**MASTER EN SCIENCES INFORMATIQUES**

# Phénomène de dégradation des performances dans un système de base de données relationnelle

Cordonnier, Vincent

*Award date:*
2013

*Awarding institution:*
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012-2013

# Phénomène de dégradation des performances dans un système de gestion de base de données relationnelle

Vincent CORDONNIER


Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Jean-Luc HAINAUT

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

# Remerciements

Tout d'abord je tiens à exprimer ma reconnaissance à Jean-Luc Hainaut pour le partage de ses connaissances, ses conseils avisés mais aussi pour le temps qu'il m'a accordé.

Je remercie également Benjamine Lurquin pour son soutien inconditionnel, sa disponibilité et les renseignements fournis tout au long de mon cursus universitaire.

Comme toujours, mes parents m'ont soutenu, et leurs nombreuses relectures et conseils m'ont encore prouvé à quel point nos liens sont forts.

A ma compagne pour la vie, celle qui a vécu au plus près de l'action, qui m'a soutenu, guidé et supporté : Merci Laura !

Enfin, aux autres qui n'ont pas été cités ci-dessus, eux qui de près ou de loin m'ont aidé et appuyé, merci !

# Table des matières

# Table des figures

# Introduction

Quel que soit le type de systèmes de base de données relationnelle, PostgreSQL, MS SQL Server, MySQL ou encore Oracle, le phénomène de dégradation des performances est une problématique bien connue des administrateurs de bases de données.

Cette entrave au bon fonctionnement des structures mises en place a déjà été décrite pour plusieurs systèmes précédemment cités.

Aujourd'hui, les systèmes informatiques sont établis dans le but d'assurer le traitement d'énormes quantités de données, et ce, de manière rapide et automatisée. Ils offrent également bon nombre de fonctionnalités : stockage durable de l'information, temps d'accès et traitements rapides d'un grand volume de données, etc.

Toutefois, de par leur complexité, ces systèmes de bases de données voient leurs performances perdre de la vitesse au fil du temps et des opérations réalisées. Ce constat est surtout marquant dans certains scénarios. Ces derniers seront recherchés et leurs performances seront analysées et quantifiées au fil de ce document.

Plusieurs facteurs vont impacter les performances au sein des systèmes de base de données. Premièrement, il existe des contraintes physiques telles que la division des ressources du système entre les différents processus démarrés au sein de celui-ci, ou encore, les temps nécessaires aux opérations de lecture et d'écriture.

Deuxièmement, des contraintes liées au choix de l'implémentation d'un système d'exploitation ou d'un système de gestion de base de données sont observées. Par exemple : les choix de mécanismes de mise en cache, ou encore la manière d'implémenter les accès concurrents.

Troisièmement, il existe des contraintes liées aux structures dans lesquelles l'information est stockée au sein d'une base de données. Ces différentes structures sont prévues afin de répondre à des besoins spécifiques (e.g. haut taux de lectures, d'insertions, de modifications, etc.). Elles sont également prévues pour être dynamiques et pour évoluer avec le temps en fonction des opérations réalisées.

Ces trois facteurs vont finalement influencer négativement l'exécution des applications qui manipulent ces données.

Si les deux premiers facteurs ont déjà été décrits dans de nombreuses recherches, les mécanismes internes d'Oracle n'ont, à ce jour, pas encore fait l'objet d'une description détaillée par ce fournisseur.

La littérature conseille bien sûr des méthodologies en fonction des cas d'utilisation choisis et donne des indications quant à l'implémentation à suivre en fonction des opérations réalisées.

Cependant, une grande zone d'ombre subsiste lorsqu'il s'agit du fonctionnement interne, propre à Oracle : gestion de blocs, stockage de l'information, type de divisions... De plus, les informations relatives à la dégradation des performances au sein des bases de données d'Oracle restent assez évasives et peu documentées. La documentation rédigée par Oracle, lui-même, ne s'attarde d'ailleurs pas sur ce phénomène.

C'est donc sur cette constatation que s'oriente le présent mémoire.

Ce travail a pour objectif d'approcher et d'analyser les différents concepts et fonctionnements propres aux systèmes de base de données fournis par Oracle afin de mettre en avant, in fine, une série de comportements qui affectent de manières différentes et significatives les performances du système.

Afin de tenter de fournir une analyse sur la dégradation des performances, une méthodologie de travail sera mise en place : plusieurs techniques spécifiques seront mises en lumière au travers d'exemples, et ce, en analysant trois structures classiques utilisées dans un système de bases de données : la table de type Heap Organized Table (*HOT*), l'index dédiée à une table *HOT* et la table de type Index Organized Table (*IOT*).

Pour compléter davantage cette étude, nous analyserons l'impact que peut avoir le phénomène de fragmentation sur le système. Dans un premier temps, nous présenterons les éléments tels que l'utilisation de la structure logique de ce système (sous certaines conditions) ou l'ordre de chargement d'une table et, dans un deuxième temps, les éléments qui participent à la perte de performances de la structure comme le chaînage et les migrations.

La dégradation du temps d'accès aux données faisant partie intégrante de la problématique, un modèle de calcul du temps d'accès aux tables sera proposé dans différents scénarios d'utilisation classique de base de données.

Une fois cette modélisation accomplie, l'ensemble des cas étudiés fera l'objet de simulations. Celles-ci permettront de constater le comportement des différentes structures énoncées et de discerner si cette modélisation correspond bien aux fonctionnements observés au sein d'Oracle.

Enfin, l'étude du comportement des structures présentées permettra, in fine, de comparer plus précisément les forces et les faiblesses de ces structures lors de leurs utilisations respectives.

Ce travail s'articule autour de huit sections, partant d'un volet théorique pour se diriger vers l'étude et l'analyse de cas spécifiques.

Le premier chapitre a pour vocation de présenter les mécanismes liés à une base de données Oracle : comment s'organise la mémoire, quels sont les processus mis en place et quels sont les différents fichiers liés à Oracle ?

Le deuxième chapitre donne une explication sur la manière dont Oracle stocke et accède aux données : quelles sont les deux types de structuration de données (structure physique et structure logique) ?

La troisième section s'attarde sur les trois types d'objets (*HOT*, *IOT* et Index) à partir desquels l'étude sera menée. Il s'agit ici de comprendre la manière dont s'organisent ces structures et dont elles sont stockées au sein des blocs.

Ensuite, le quatrième chapitre tente d'apporter des éléments de réponses sur l'origine de la baisse de performance des systèmes de gestion de bases de données, en posant les bases du principe de fragmentation.

Sur ces quatre sections s'achèvent le volet théorique de ce mémoire pour passer à la mise en place des modèles mathématiques permettant de calculer le coût d'accès à l'information.

La cinquième section a donc pour objectif de modéliser le phénomène de fragmentation.

Le chapitre six, quant à lui, met en exergue la structure d'une table *HOT* et de son index afin d'étudier celle-ci au travers de différents cas pratiques dans le but d'identifier les éléments participant à la perte de performances de cette structure.

La section suivante réalise la même étude, mais cette fois sur la structure d'une table *IOT* afin d'identifier également les éléments qui auront un impact sur la qualité des performances de cette table.

Si l'ensemble des sections précédemment exposées permet de construire un modèle mathématique et de l'analyser, la dernière section a pour objectif de synthétiser tous les résultats collectés afin d'avoir une vision globale des performances sous certaines hypothèses.

Enfin, ce travail se clôturera par une conclusion qui traduira les résultats découverts tout au long de ce document.

# 1 Présentation d'Oracle

## 1.1 Préambule

A ce jour, de nombreux Systèmes de Gestion de Base de Données (*SGBD*) sont disponibles sur le marché. Chacun de ces systèmes possèdent une organisation et des modes de fonctionnement qui leur sont propres. Qu'est-ce qu'Oracle et quelle est sa position sur le marché des *SGBD* ? Quels sont les mécanismes liés à un *SGBD* Oracle et comment organise-t-il sa mémoire afin d'augmenter son efficience ? Quels processus spécifiques met-il en place afin d'augmenter ses performances et quels sont leurs rôles ?

L'ensemble de cette section s'appuie sur les développements énoncés dans la documentation en ligne *Oracle Database Administrator's Guide* [12] ainsi que dans les livres *Expert Oracle Database Architecture, Oracle Database 9i, 10g and 11g Programming Techniques and Solutions* [1] et *Oracle Performance Survival Guide* [15].

## 1.2 Qu'est-ce qu'Oracle ?

Oracle est un *SGBD* relationnel objet fourni par Oracle Corporation. Ce *SGBD* est actuellement un des leaders du marché des bases de données. Il existe plusieurs gammes du produit dont les deux principales sont la version « Standard Edition » et la version « Enterprise Edition ». Oracle fournit également plusieurs options payantes permettant des fonctionnalités avancées telles qu'OLAP, Advanced Security, le partionning, etc.

Oracle se base sur le langage procédural *SQL* et les applications exécutent des requêtes *SQLPL/SQL* (programming language *SQL*) afin de réaliser des actions sur le *SGBD*.

Ce moteur est composé d'un noyau qui garantit les propriétés de confidentialité, d'intégrité et de cohérence du système. Il gère également les accès concurrents, optimise les requêtes ou s'occupe de la gestion du stockage physique des données. Il est également composé d'un dictionnaire de données dans lequel l'ensemble de la structure du *SGBD* est gérée suivant un schéma relationnel. Cette structure permet de décrire l'ensemble des objets au sein d'Oracle de manière dynamique. Le langage *SQL* constitue une interface entre les outils qui se connectent à Oracle et son noyau.

## 1.3 Organisation de la mémoire

Les lignes suivantes présentent le fonctionnement global de la mémoire d'Oracle. Le moteur Oracle repose sur des zones de mémoires distinctes. Lors d'une connexion, un client interagit avec ces différentes mémoires. Il existe deux grandes divisions de mémoire :

- d'une part la *System Global Area* (*SGA*) qui est une zone mémoire réservée au système et qui contient la *Shared Pool* (*SP*) et la *Large Pool* (*LP*) ;
- d'autre part, la *Process Global Area* (*PGA*) et la *User Global Area* (*UGA*) qui contiennent les données des sessions ouvertes.

La Figure 1-1 illustre l'utilisation d'une base de données Oracle à laquelle trois clients se sont connectés en mode dédié (cf. 1.4.1 ). Cette représentation met en avant la séparation des différentes mémoires sur lesquelles se base le moteur Oracle.

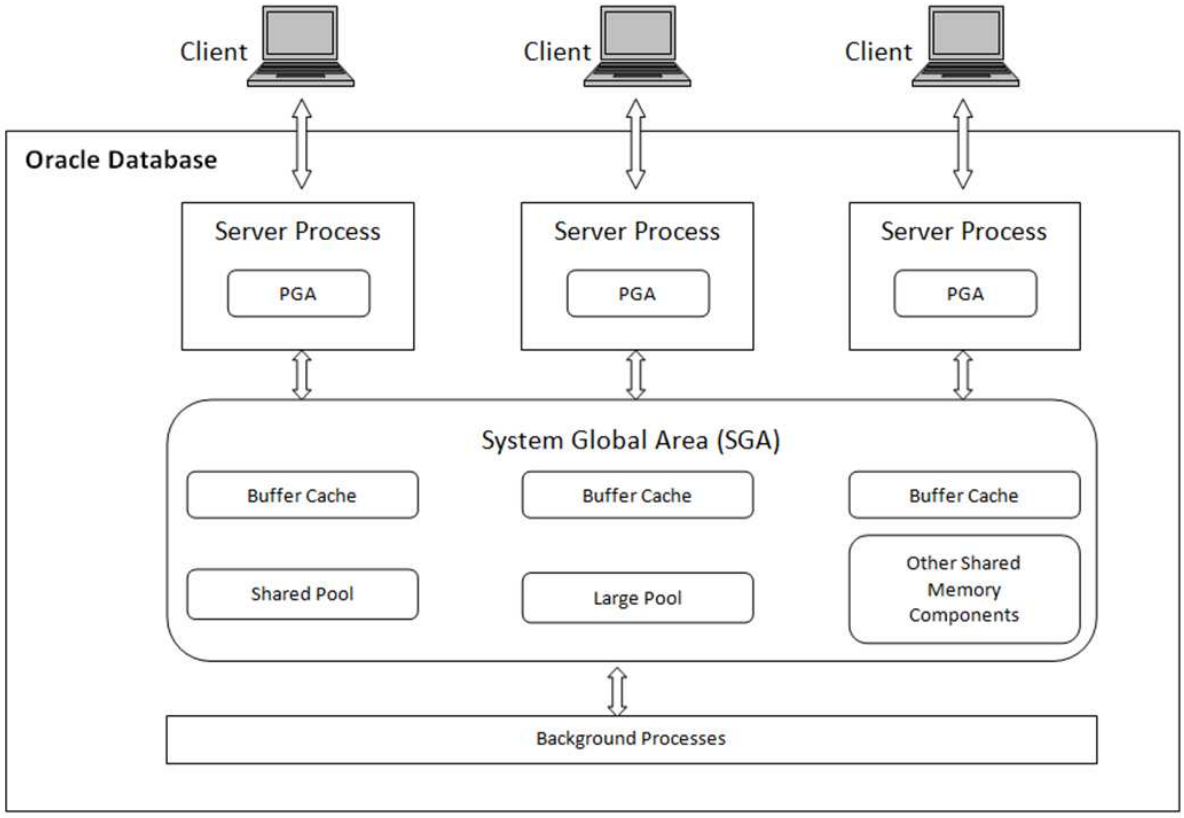


**Figure 1-1 : Organisation de la mémoire d'Oracle lors de connexions dédiées**

### 1.3.1 System Global Area

La *System Global Area* (*SGA*) est une zone mémoire réservée au moteur Oracle. Cette mémoire est divisée en différentes parties utilisées par un ensemble de fonctionnalités :

- Charger les dictionnaires de données contenant la structure des tables afin de permettre aux processus un accès rapide à ces données (les métadonnées) ;
- Stocker des informations sur les transactions pour lesquelles une validation a été donnée, mais qui ne sont pas encore écrites physiquement dans le *Redo Log* (cf. section 1.5.2 ). La *SGA* est utilisée tel un *buffer* permettant de s'abstraire de l'écriture physique des *logs* ;
- Stocker les informations lues depuis un fichier de données dans une mémoire cache. Il s'agit typiquement de charger les données contenues dans un fichier de paramètres qui configurent le fonctionnement du moteur Oracle ;
- Contenir la *Shared Pool* (*SP*) qui contient les informations relatives au fonctionnement interne d'Oracle ;
- Contenir la *Large Pool* (*LP*) qui contient le reste des informations relatives au fonctionnement interne d'Oracle. Celle-ci est utilisée pour soulager la mémoire *SP*.

Un grand nombre d'actions sont réalisées au sein de la *SGA*. La performance du moteur Oracle est fortement dépendante de la taille allouée à la *SGA*.

Oracle utilise cette mémoire comme tampon pour éviter d'effectuer des opérations en mémoire plutôt que sur un disque dur. La taille allouée à la SGA ainsi que sa vitesse affecte grandement les performances d'Oracle.

### 1.3.2 Database Buffer Cache

La mémoire tampon de la base de données Oracle est un des composants les plus importants au sein de la *SGA*. Il s'agit d'une zone de mémoire qui est allouée pour stocker les blocs de données (cf. section 2.2.2 ) lors des manipulations *SQL*.

Cette mémoire est partagée entre tous les processus du moteur Oracle. Plus particulièrement, l'ensemble des blocs chargés dans la mémoire tampon est disponible pour l'ensemble des sessions.

Cette mémoire est composée de deux listes : la liste des blocs à écrire et la liste des derniers blocs utilisés.

La première liste contient l'ensemble des blocs qui ont été modifiés et qui doivent être réécrit au sein des fichiers de base de données (cf. section 1.5.2 ) par le *Database Block Writer* (*DBWn*) (cf. section 1.4.2 ).

La seconde liste *Least Recent Used* (*LRU*) *List* est la mémoire qui contient l'ensemble des blocs qui peuvent être utilisés comme tampon. Celle-ci est scindé en plusieurs catégories : *Pinned*, *Clean*, *Free* ou *Unused* et, enfin, *Dirty buffers*. La catégorie *Pinned* désigne la mémoire tampon qui est actuellement en cours d'utilisation. La catégorie *Clean* désigne les zones du tampon qui sont prêtes à l'usage. La catégorie *Free* ou *Unused* désigne une zone de la mémoire tampon qui n'a pas encore été utilisée. La catégorie *Dirty buffers* est composée d'éléments qui viennent de la liste à écrire.

Lorsqu'un processus Oracle nécessite un bloc de données spécifiques, il commence par chercher dans cette mémoire tampon. Si le bloc souhaité est disponible, il y accède directement. Si la recherche ne donne aucun résultat, alors les données sont chargées depuis le fichier de données se trouvant sur le disque. Après la lecture du bloc dans le fichier, les données de celui-ci sont transférées vers un *Free buffer*.

Lors du chargement d'un bloc, s'il ne reste plus d'espace disponible, une recherche est réalisée afin de trouver de la mémoire de type *Free* ou *Unused.* Lors de ce processus de recherche, si un *Dirty buffers* est détecté, il est déplacé vers la liste de blocs à écrire.

En règle générale, les zones du tampon lues sont placées à la fin de la liste *LRU*. Ce mécanisme permet de garder les blocs les plus récemment utilisés, le plus longtemps dans la *LRU*. Lors d'un *full table scan*, les blocs lus sont placés au début de la liste et seront donc les premiers candidats au nettoyage.

### 1.3.3 Shared Pool

La *Share Pool* (*SP*) est une mémoire qui représente un des éléments les plus critiques de la *SGA*. C'est au sein de la *SP* qu'Oracle stocke les *query plans SQL* ou *SQLPL/SQL* déjà analysés. De plus, les dictionnaires de données sont également stockés dans la *SP*. En résumé, l'ensemble des éléments relatifs au fonctionnement interne d'Oracle est placé en cache dans la *SP*. Comme son nom l'indique, cette mémoire est utilisée par l'ensemble des utilisateurs de la base de données. La *SP* sert de cache et retient un maximum de valeurs utiles parmi celles déjà utilisées depuis le démarrage du système, afin d'optimiser les requêtes à venir.

### 1.3.4 Large Pool

La *Large Pool* (*LP*) est une zone mémoire qui contient également des informations internes au moteur Oracle. Elle prend en charge les éléments qui ne sont pas directement liés à l'optimisation du cache. En effet, la *SP* n'est pas conçue de manière à supporter ces éléments. La *LP* a trois objectifs définis :

- Fournir aux connexions partagées un espace pour allouer les *User Global Area* (*UGA*) ;
- Stocker des informations permettant de coordonner les exécutions parallèles de requêtes;
- Servir de cache aux processus de prise de *backup*.

### 1.3.5 Process Global Area

La *Process Global Area* (*PGA*) est une zone de mémoire dédiée à un processus. De manière générale, chaque connexion en mode dédié reçoit une zone mémoire nommée *PGA,* utilisée comme cache. La *PGA* évite de surcharger la *SGA* et est vidée une fois la session clôturée.

### 1.3.6 User Global Area

La *User Global Area* (*UGA*) contient toutes les données utilisées par l'ensemble des sessions lorsqu'une connexion en mode dédié est réalisée. Cette zone mémoire est accessible par tout le groupe de serveurs partagés. Ainsi, l'ensemble des utilisateurs dispose, potentiellement, de mémoire supplémentaire partagée. Celle-ci doit toutefois être répartie afin de permettre l'utilisation concurrente.

### 1.3.7 Autres mémoires

D'autres mémoires existent au sein d'Oracle telles que la *Java Pool* ou la *Streams Pool*. Elles sont utilisées dans certains cas spécifiques tel que la manipulation d'objets java ou la manipulation de queues de messages. Ce présent travail ne les décrit pas et elles ne sont, dès lors, pas abordées.

## 1.4 Processus Oracle

Cette section décrit les processus majeurs qui composent le moteur Oracle.

### 1.4.1 Processus serveur

Oracle peut gérer les connexions en suivant deux modes: le mode dédié (*Dedicated Server*) et le mode partagé (*Shared Server*).

**Connexion dédiée**

Dans ce mode, Oracle crée un processus dédié afin de répondre aux demandes d'une connexion jusqu'à la clôture de celle-ci. La Figure 1-2 illustre ce fonctionnement. Lorsqu'une session est ouverte, puis est laissée en attente, elle monopolise inutilement des ressources du système. Plusieurs sessions peuvent travailler sur les mêmes données, la *SGA* se charge de conserver la consistance du système.

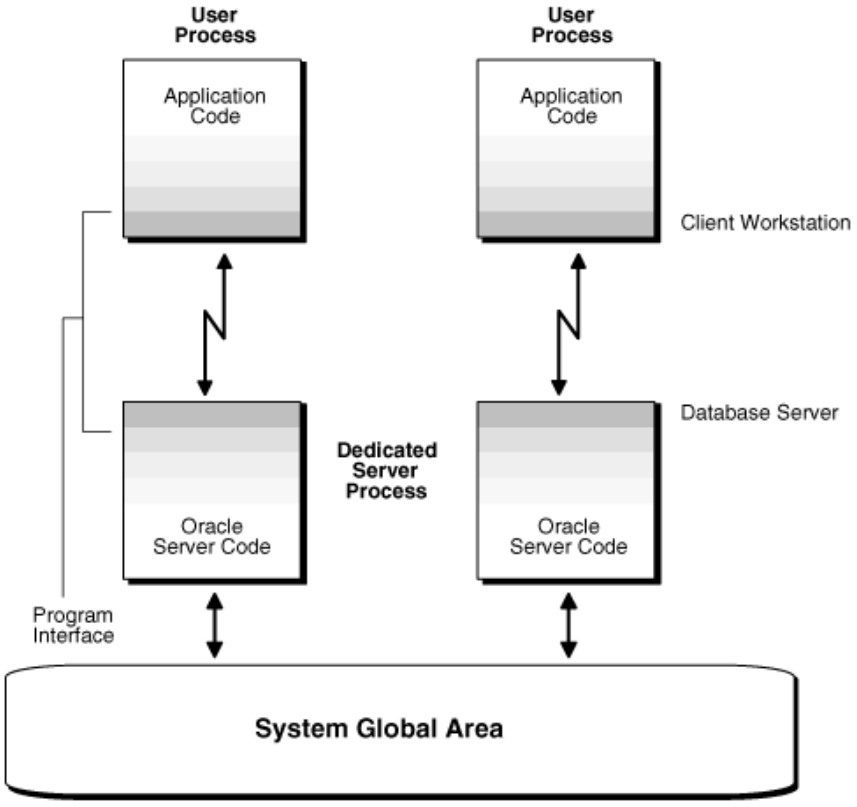


**Figure 1-2 : Organisation de deux connexions dédiées [11]**

**Connexion partagée**

À ce moment, un distributeur est mis en place afin de répartir les ressources à chaque demande d'un client. Dans ce mode, l'*UGA* est partagée par l'ensemble des processus serveurs. La Figure 1-3 illustre ce mode de fonctionnement qui permet d'optimiser l'utilisation des ressources. En raison du rôle central du distributeur, en cas de surcharge liée à la gestion des processus, le nombre de transactions concurrentes exécutées à la seconde a tendance à diminuer si le nombre d'utilisateurs augmente au-delà de la capacité du serveur.

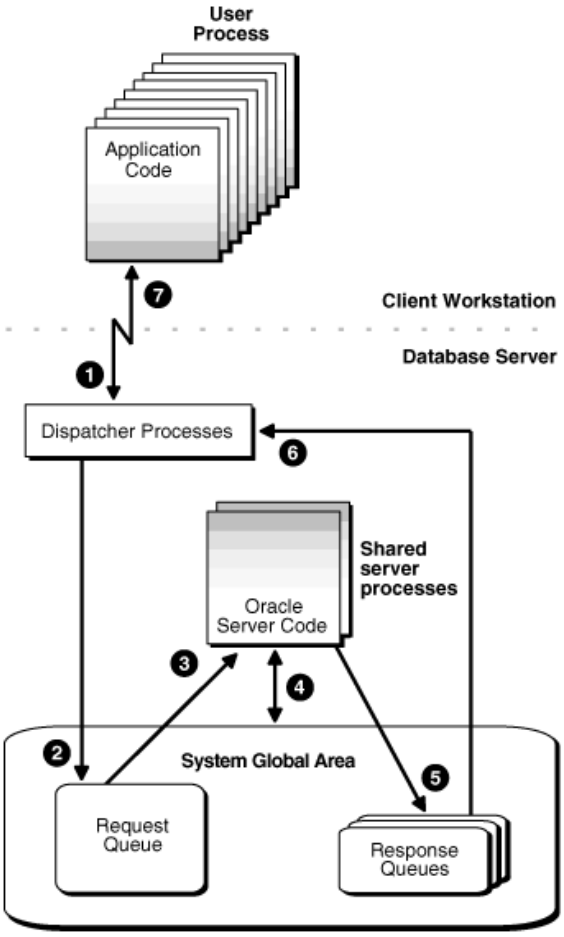


**Figure 1-3 : Organisation d'un "Shared server" [11]**

### 1.4.2 Processus de fond

Il existe de nombreux processus au sein d'Oracle. Deux d'entre eux sont plus critiques. Il s'agit du *moniteur de processus* (*PMON*) et du *moniteur système* (*SMON*). Ils sont mis en place afin de coordonner les autres processus au sein d'Oracle. La Figure 1-4 schématise les interactions entre ces différents processus. Les lignes qui suivent décrivent, en quelques mots, l'usage de ceux-ci ainsi que leurs rôles spécifiques.

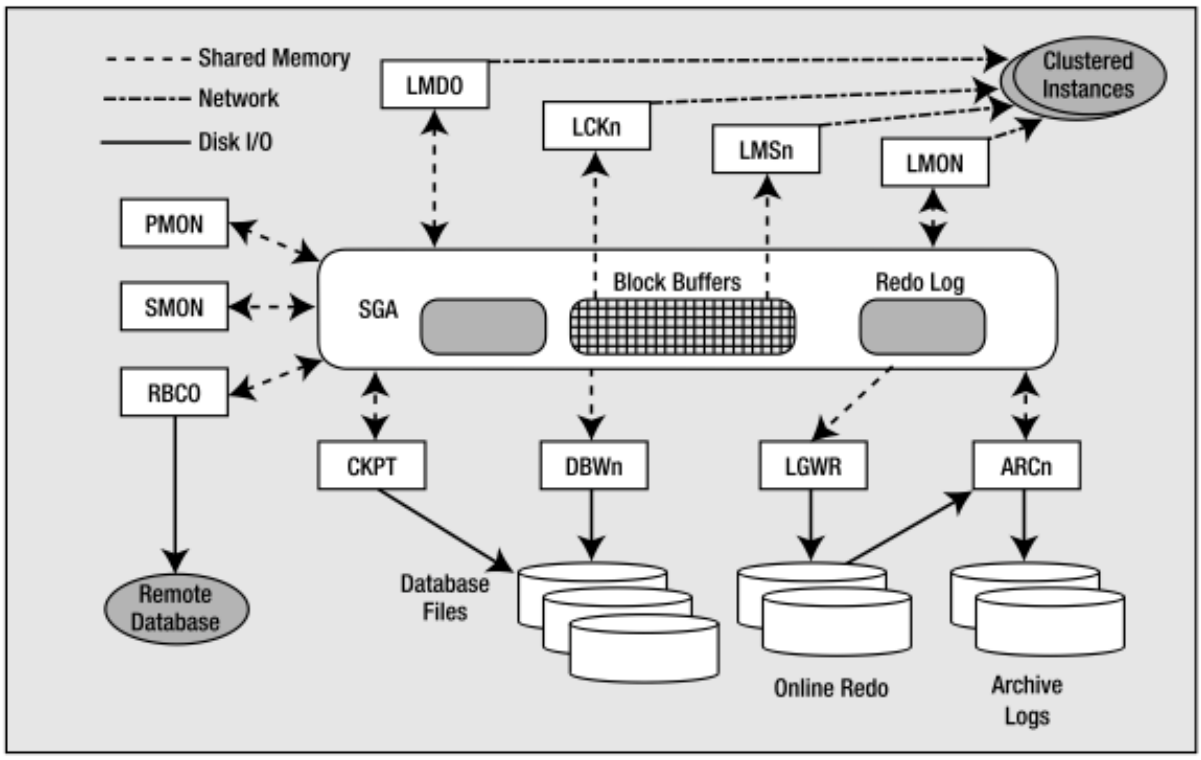


**Figure 1-4 : Observation des Processus de fond [1]**

### Moniteur de processus

La première tâche du *PMON* est un rôle de nettoyage orienté processus client. En effet, il va chercher toutes les connexions terminées de façon anormale, et effectue un retour en arrière des transactions pour lesquelles une validation n'a pas été accordée. De plus, il garantit le nettoyage de la *SGA* des ressources utilisées par cette ancienne connexion. Enfin, le *PMON* a aussi un rôle vis-à-vis des autres processus de fond : il vérifie que ces derniers sont actifs, et, dans le cas contraire, il les redémarre.

### Moniteur système

Le *SMON* est en charge du bon fonctionnement du système via un ensemble de tâches. Il prend en charge :

- Le nettoyage des *tablespaces* temporaires ;
- L'obtention de la prochaine extension contigüe au niveau du système de fichier lorsqu'il faut agrandir un *tablespace* ;
- La récupération des transactions étant dans un statut anormal dans le fichier (e.g. suite à un redémarrage non désiré du moteur Oracle) ;
- Dans une architecture de *haute disponibilité* (*HA*), lorsqu'une erreur survient, il prend en charge la récupération de l'ensemble des éléments de l'instance qui étaient et sont encore consistants ;
- Le nettoyage des dictionnaires d'objets lors de la suppression de l'un d'eux.

**Autres processus**

Il existe plusieurs autres processus utilisés par Oracle. En voici un aperçu :

- *Distributed Database Recovery* (*RECO*) : gère le *two phase commit* lors d'une transaction distribuée entre différentes bases de données ;
- *Checkpoint Process* (*CKPT*) : met à jour l'en-tête des fichiers ;
- *Database Block Writer* (*DBWn*) : est chargé d'écrire le contenu du *buffer* sur les disques afin de libérer la mémoire cache ;
- *Log Writer* (*LGWR*) : est chargé de nettoyer le « *REDO Log buffer* » en réécrivant son contenu dans les « *REDO Files* » ;
- *Archive Process* (*ARCn*) : est chargé de copier le contenu du « *REDO File* » actif vers un fichier archive ;
- *Diagnosability Process* (*DIAG*) : est chargé de surveiller l'état général de l'instance Oracle ;
- *Flashback Data Archiver Process* (*FBDA*) : est chargé de lire l'*UNDO log* afin d'obtenir un enregistrement à un moment X (cette valeur représentant un moment dans le passé) ;
- *Database Resource Manager Process* (*DBRM*) : est chargé de gérer la quantité de ressources disponibles pour différentes opérations ;
- *General Task Execution Process* (*GENO*) : est chargé de la gestion interprocessus. Si un de ceux-ci est potentiellement dangereux (*e.g.,* si l'exécution du processus A risque de stopper le processus B), alors *GENO* place ce processus en tâche de fond.

Oracle utilise encore d'autres processus afin de fournir l'ensemble de son catalogue de services. La documentation d'Oracle référence correctement ceux-ci. En cas de nécessité, il est intéressant de s'y rapporter [11].

## 1.5 Les fichiers d'Oracle

Cette section décrit les différents fichiers liés à Oracle. Pour être complet, les fichiers systèmes ainsi que les fichiers propres à certains mécanismes de récupération d'Oracle sont également présentés. Afin de se concentrer sur le sujet de ce mémoire, une partie plus détaillée est dédiée aux fichiers contenant les données du *SGBD*.

### 1.5.1 Fichiers systèmes

Les fichiers systèmes, d'une part, définissent une série d'options qu'Oracle utilise lors de son démarrage ; d'autre part, permettent de vérifier le bon fonctionnement d'Oracle et de diagnostiquer d'éventuelles erreurs qui peuvent survenir lors de l'utilisation du *SGBD*.

**Fichiers de paramètres**

Ces fichiers définissent les options de démarrage d'Oracle. Ceci inclut la taille des différentes mémoires ou, encore, la localisation des fichiers nécessaires au bon fonctionnement d'Oracle. Une partie des paramètres peuvent être changés « à chaud » une fois le *SGBD* démarré, tandis que d'autres exigent d'être définis avant le démarrage d'Oracle.

#### Fichiers de traces

Ces fichiers sont générés par les processus Oracle afin de permettre le diagnostic des évènements imprévus. Ces fichiers contiennent des erreurs spécifiquement liées au fonctionnement du *SGDB*.

#### Fichiers d'alertes

Ces fichiers contiennent les erreurs générées et identifiées lors de l'utilisation des bases de données. Les alertes reprennent, entre autres, l'ensemble des erreurs syntaxiques et les erreurs de droits lors de tentatives d'accès à un objet.

### 1.5.2 Fichiers de la base de données

Ces fichiers contiennent les données. Dans chacun de ces fichiers, une zone est réservée afin de stocker des métadonnées, et le reste du fichier contient la structure logique et les données qui lui sont associées. Il existe quatre différents groupes:

- Les fichiers de données qui contiennent les tables, les index ou tout autre type d'informations pouvant être contenues dans un segment ;
- Les fichiers de données temporaires, sensiblement identiques aux fichiers de données, sont utilisés pour stocker l'information de manière provisoire ou pour réaliser les tris ;
- Les fichiers de contrôle qui contiennent les informations permettant de lier un fichier avec ses métadonnées. Les liens entre les fichiers de *backup* et les fichiers de données se trouvent également à l'intérieur de ce type de fichiers ;
- Les fichiers de *log* (en Anglais « *Redo Log »*) contiennent les piles d'instructions liées à une transaction. Les lignes suivantes décrivent ces fichiers plus précisément.

#### Stockage des données

Les données sont stockées dans les conteneurs de données. Une fois qu'Oracle est mis à l'arrêt, les données sont stockées dans ces fichiers particuliers : il s'agit des fichiers les plus importants avec les fichiers de *log* (cf. section 1.5.2 ). Il existe différents types de conteneurs de données :

- Les fichiers : ceux-ci peuvent être listés ou même déplacés via des commandes classiques du *système d'exploitation* (*SE*) ;
- Les partitions : ces dernières sont allouées à Oracle et le *SE* n'a pas d'accès à leur contenu;
- Le gestionnaire de fichiers propre à Oracle : *Automatic Storage Management* (*ASM*). L'*ASM* permet de stocker les informations au sein d'une base de données relationnelle ;
- Le système de fichiers partitionné et partagé au sein d'un environnement entre différentes machines.

#### Fichiers temporaires

Les fichiers temporaires représentent un type de fichiers de données ayant une utilisation particulière. Oracle utilise cette zone temporaire afin d'accomplir les opérations ne pouvant être réalisées en mémoire vive. Il s'agit donc d'opérations temporaires généralement exécutées sur un nombre de données assez important. Ces fichiers stockent uniquement des données temporaires (Tables ou Index) ou des résultats intermédiaires (e.g. un tri de données), avant qu'ils ne soient utilisés.

Les fichiers temporaires disposent de plusieurs avantages. D'abord, toutes les opérations effectuées sur ces derniers ne sont pas stockées dans les fichiers de *log*, limitant ainsi le nombre de lectures et d'écritures. Ensuite, ces fichiers temporaires peuvent être partagés entre différents *tablespaces*. Une seule zone tampon pour l'ensemble des processus permet d'économiser de l'espace. Enfin, l'utilisation de cet espace évite de polluer les *tablespaces* contenant les données.

#### Fichiers de contrôle

Ces fichiers référencent l'ensemble des fichiers dont Oracle a besoin afin d'accéder correctement aux données. Ils contiennent des informations liant les données à leurs fichiers, aux points de contrôles réalisés (*checkpoints*), ainsi qu'à leur base de données, leurs archives et enfin leurs *backups*. Ces fichiers sont très importants pour le *DataBase Administrator* (*DBA*) car ils permettent de faire le lien entre les différents fichiers et concepts. Ils peuvent être répliqués à plusieurs endroits différents.

#### Fichiers de log

Ces fichiers constituent un autre élément très important au sein d'Oracle. Ils stockent les actions lors d'une transaction au sein des bases de données. Ils sont utiles afin de maintenir et garantir l'intégrité des données lors du *crash* d'une base de données ou encore, lors d'un simple retour en arrière (*rollback*). En cas de *crash*, les *REDO* sont lus, et Oracle applique toutes les transactions contenues dans les *REDO* non encore exécutées, sur les fichiers de données. Généralement, les *REDO* sont composés de plusieurs fichiers (par exemple : *REDO1, REDO2 et REDO3*) utilisés alternativement. Un seul de ces fichiers est actif à un moment donné. En effet, lorsque des transactions sont réalisées, le *Log writer* (*LGWR*) vide le *Redo Buffer* vers *REDO* actif. Une fois le fichier rempli, Oracle réalise un *log switch* et entame l'écriture sur *REDO2*. Comme le *REDO1* n'est plus utilisé, un processus peut être démarré afin de vider *REDO1* vers une zone de *backup*. Le contenu de *REDO1* est alors copié vers un fichier d'archive afin que *REDO1* puisse être réutilisable aussi vite que possible.

Comme l'entièreté des opérations de modification des données est stockée au sein des *Redo Log*, et que ces fichiers sont copiés vers les archives avant d'être réutilisés, il est possible de revenir en arrière, à un état consistant, en lisant à l'envers les *Redo Log* (rétention courte) ou en reparcourant les archives (rétention aussi grande que les disques le permettent).

### 1.5.3 Recovery Files

#### Change-tracking Files

Les *Change-tracking Files* contiennent l'ensemble des blocs modifiés depuis le dernier *backup* incrémental. De cette manière, le *Recovery Manager* (*RMAN*), mécanisme de récupération des données d'Oracle, peut prendre en *backup* uniquement les blocs qui ont été modifiés sans avoir à vérifier l'ensemble de la base de données. Ce mécanisme est très efficient lorsque les données ne sont pas modifiées au sein de la majorité des blocs qui composent le fichier.

#### Flashback log Files

Les *Flashback log Files* sont utilisés afin de permettre d'accéder aux données telles qu'elles étaient à un temps X, sans toutefois devoir restaurer la base de données. Plus ces fichiers sont conséquents et plus il est possible de remonter dans le temps.

## 1.6 Synthèse

Cette section présente Oracle de manière globale. Elle décrit la position d'Oracle en tant qu'acteur sur le marché des *SGBD*. Puis elle énonce et définit les composants qui constituent le moteur Oracle.

Le mode de fonctionnement de ce dernier peut être de deux types : les ressources sont soit dédiées pour chaque connexion, soit, partagées entre l'ensemble des connexions.

Ces ressources se divisent en deux catégories principales :

- les processus qui sont les éléments permettant au système de fournir les fonctionnalités attendues d'un *SGBD* ;
- la mémoire qui fournit des mécanismes de cache performants.

Le moteur Oracle se base principalement sur deux zones de mémoires distinctes (SGA et PGA) qui lui permettent de maximiser ses performances.

Pour assurer son bon fonctionnement, Oracle s'appuie également sur une série de fichiers systèmes. De plus, il stocke les informations business au sein des fichiers de données. Enfin, un dernier type de fichiers est utilisé afin de garantir les mécanismes de récupération.

# 2 Organisation des données

## 2.1 Préambule

La section précédente définit les différents composants qui constituent Oracle. Comme pour chaque SGDB, les performances du moteur Oracle sont liées à la manière dont celui-ci peut stocker et accéder aux données. Comment Oracle stocke-t-il ces données, tant au niveau physique que logique et comment organise-t-il les enregistrements au sein des deux niveaux?

L'ensemble de cette section s'appuie sur les développements énoncés dans la documentation en ligne *Oracle Database Administrator's Guide* [12] ainsi que dans le livre *Expert Oracle Database Architecture, Oracle Database 9i, 10g and 11g Programming Techniques and Solutions* [1].

## 2.2 Présentation des conteneurs de données

Cette section décrit les conteneurs de données utilisés au sein d'Oracle. Ce dernier organise ses données en deux types : la structure logique et la structure physique.

La Figure 2-1 représente le lien entre ces deux structures. Le système d'exploitation est en charge de la structure physique. Le moteur Oracle garantit la structure logique ainsi que le lien, à l'aide d'identifiants, entre ces deux structures.

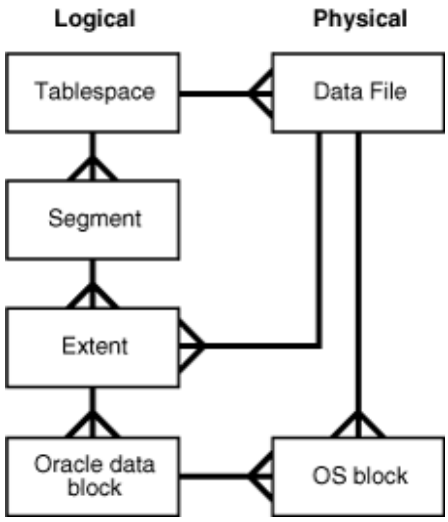


**Figure 2-1 : Présentation de l'organisation des données [11]**

### 2.2.1 La structure physique

Il s'agit ici de décrire la manière dont le *SGBD* organise ses données de manière physique avec l'aide du *SE*. Cette analyse ne décrit pas le matériel physique qui héberge les données. La structure logique est quant à elle détaillée.

**Data files**

Les fichiers de données sont constitués d'une suite de blocs physiques. Ces fichiers se placent dans un dossier qui contient l'ensemble de ceux-ci. Les fichiers se composent d'un en-tête qui fournit diverses métadonnées telles que les changements validés ou encore la taille du fichier et des sections contenant les données. Chaque fichier est identifié de manière unique au sein du système.

**Blocs du système d'exploitation**

Les blocs du *SE* correspondent aux blocs physiques fournis par le système.

### 2.2.2 La structure logique

Oracle se base sur une cascade de conteneurs afin d'organiser ses données. La Figure 2-2 illustre la structure de cette organisation. Les prochains développements décrivent plus précisément les différentes relations entre ces conteneurs.

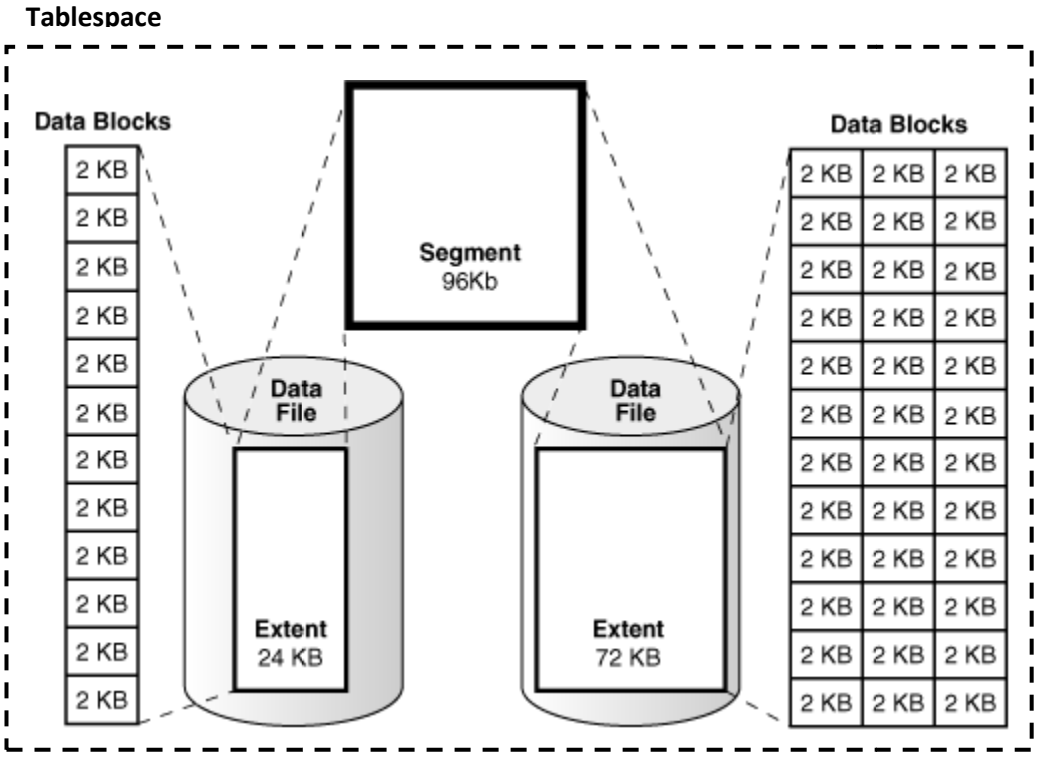


**Figure 2-2 : Exemple d'ordonnancement logique [11]**

**Tablespace**

Le *tablespace* est la plus grande unité de stockage logique. Il est composé de segments. Au niveau physique, il est lié à un ou plusieurs fichiers de données. Une fois ce conteneur créé et associé à un ensemble de fichiers, les utilisateurs peuvent le manipuler sans connaître sa structure au niveau des fichiers sur le disque.

**Segment**

[1] Le segment est un conteneur qui contient toutes les données pour une structure logique spécifique. En d'autres termes, lors de la création d'un conteneur, telle une table, un nouveau segment est créé et reçoit une structure logique. Un segment est constitué par l'ensemble des *extent(s)* ou extension(s) qu'il a reçu lors de chaque agrandissement de sa structure. Chaque création d'une structure logique au sein d'un *tablespace* se nomme « objet ». Celui-ci est de différents types:

- *CLUSTER* : Ce type de segment héberge des tables. En Oracle, il existe deux types de tables clusters: la table *B-Tree* et la table *hash*. Les *clusters* sont communément utilisés afin de stocker des données connexes provenant de plusieurs tables pré-jointes au sein du même

bloc. Le terme *cluster* se réfère à la capacité de ce segment à regrouper, au sein du même emplacement physique, des informations reliées entre elles ;

- *INDEX* : Ce type de segment héberge la structure de type index ;
- *INDEX PARTITION* : Ce type de segment contient un index des données comprises dans une table partitionnée ;
- *LOB PARTITION* : Lorsqu'une table partitionnée contient un large objet, ce dernier sera stocké dans ce type de segment;
- *LOBINDEX* : Ce type de segment contient la structure d'un large objet (composé d'une série de *pointer* permettant de constituer le *LOB* contenu au sein d'un *LOB SE*GMENT) ;
- *LOBSEGMENT* : Ce type de segment englobe les données contenues dans un large objet. Une table qui contient une colonne de type *LOB* placera le contenu de cette colonne dans un segment de ce type ;
- *NESTED TABLE* : Ce type de segment contient les données de tables imbriquées. Oracle permet de stocker une table au sein d'une ligne avec les tables imbriquées ;
- *ROLLBACK* : Ce type de segment permet de stocker les données permettant les retours en arrière. Il est mis en place explicitement par un *DBA* ;
- *TABLE* : Ce type de segment contient les données d'une table au sein de la base de données ;
- *TABLE PARTITION* : Ce type de segment contient une partie des données d'une table partitionnée. Cette table est constituée d'un ou plusieurs segments de ce type ;
- *TYPE2 UNDO* : Ce type de segment permet de stocker les données permettant les retours en arrière. Il est géré uniquement par le moteur Oracle.

**Extent**

Un *extent* ou extension est un groupe de blocs contigus alloués à un segment afin de stocker un type d'information. Lors d'ajouts de données dans un objet, le moteur Oracle cherche un espace disponible pouvant contenir les nouvelles informations au sein du segment. S'il ne trouve pas un espace contigu dans lequel il peut placer l'information, le moteur Oracle va allouer une ou plusieurs extensions au segment afin de pouvoir ajouter l'information. Il est important de noter que cette manière de travailler évite de fragmenter les données lors de l'insertion.

**Bloc**

Le bloc est la plus petite unité logique d'espace utilisé par Oracle (N.B., il peut être associé au terme *page* dans d'autres *SGBD* tel que MS *SQL* Server). Par défaut, un bloc a une taille de 8 Ko et doit être comprise entre 2 Ko et 32 Ko. Il doit être un multiple de la taille des blocs du système d'exploitation. Lorsqu'une base de données demande un bloc de données logique, le *SE* lui fournit un espace de données qu'il peut utiliser pour stocker ses informations. Cette méthode de séparation entre un bloc physique et un bloc logique utilisée au sein d'Oracle implique deux choses :

- L'application fait abstraction de l'adresse physique des données sur le disque ;

- Les données contenues dans une base de données peuvent être découpées sur plusieurs disques ou être copiées en « miroir » sans aucun impact sur la logique de présentation des données.

La structure d'un bloc se présente telle qu'illustrée dans la Figure 2-3. Il se compose d'un en-tête et de données. L'en-tête est lui-même composé de trois éléments :

- *Header* : les informations générales concernant ce bloc, tels que l'adresse du bloc et son type ;
- *Table Directory* : des informations sur la table qui contient ce bloc ;
- *Row Directory* : des informations sur les lignes stockées au sein de ce bloc.

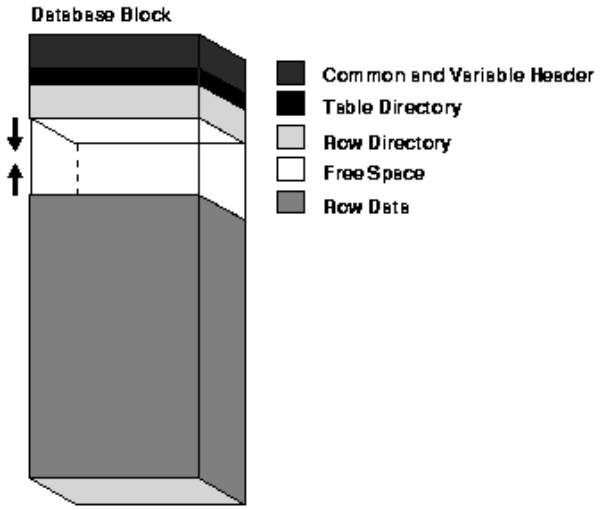


**Figure 2-3 : Représentation d'un bloc logique [11]**

Lorsque le moteur Oracle doit lire une information au sein d'un bloc, il doit charger l'entièreté du bloc en mémoire afin de pouvoir le lire [13].

Il existe deux paramètres permettant de définir les seuils d'utilisation d'un bloc :

- *PCTFREE* : Au niveau d'une table *HOT* (cf. section 3.2 ), cette valeur est le pourcentage qui définit la taille réservée pour la mise à jour des données contenues dans le bloc courant. Par exemple, si la valeur est 20, cela signifie qu'il est possible d'effectuer des insertions tant que le bloc n'atteint pas un taux de remplissage de 80%. Une fois ce taux atteint, il n'est plus possible d'ajouter une ligne dans ce bloc. Au niveau d'un index ou d'une table *IOT* (cf. sections 3.3 et 3.4 ), cette valeur est l'espace réservé pour les mises à jour après la reconstruction ;
- *PCTUSED* : Ce paramètre est utilisé afin de savoir s'il existe de l'espace utilisable dans le bloc courant. À cet effet, à la suite d'une opération de suppression ou de modification, Oracle vérifie si l'espace utilisé dans le bloc est inférieur à ((Taille du bloc - entête) * PCTUSED/100). Cette formule est issue de l'analyse fournie par Burleson Consulting [13]. Le cas échéant, le moteur du *SGBD* enregistre que le bloc peut à nouveau recevoir de l'espace. La somme de *PCTUSED* et *PCTFREE* doit être inférieure à 100. Dans le cas contraire, si

*PCTUSED* + *PCTFREE* = 100, cela peut entraîner une utilisation de l'espace du bloc en dents de scie, ce qui affecte négativement les performances.

Ces deux valeurs combinées garantissent un espace suffisant pour effectuer les mises à jour au sein du bloc (*PCTFREE*). Elles garantissent également une marge de fonctionnement : une fois le seuil atteint (*PCTFREE*), le bloc doit dégager de l'espace jusqu'à atteindre le seuil inférieur (*PCTUSED*) avant de pouvoir recommencer à insérer de nouvelles lignes de données.

## 2.3 Synthèse

L'objectif de cette section vise à comprendre le lien établi entre la structure physique des données au sein des fichiers et la structure logique utilisée par Oracle afin d'organiser l'information.

La structure physique, c'est-à-dire les fichiers et les blocs qui les composent, est propre au système d'exploitation et est utilisée pour stocker la structure logique mise en place dans le *SGBD*.

La structure logique est, quant à elle, divisée en plusieurs conteneurs logiques de tailles et de types différents. Chacun possède ses spécificités et permet à Oracle de travailler à différents niveaux de la structure de données.

Enfin, les données sont stockées dans un bloc logique, la plus petite entité de stockage, dont la structure et l'utilisation sont décrites au travers de cette section.

# 3 Les types d'objets

## 3.1 Préambule

Dans la section précédente, le lien établi entre la structure physique et la structure logique permet de comprendre la manière dont les différents types d'objets sont stockés au sein des fichiers.

De manière générale, ce mémoire analyse la dégradation des performances de trois types d'objets disponibles au sein d'Oracle : la structure *HOT*, la structure de l'index de cette dernière et la structure *IOT*. Avant d'aller plus loin, il est important de bien comprendre le fonctionnement de ces objets.

Comment s'organisent ces trois structures et comment sont-elles stockées au sein des blocs ? Quelles sont les opérations possibles à réaliser sur celles-ci et quels sont leurs avantages et inconvénients ?

L'ensemble de cette section s'appuie sur les développements énoncés dans la documentation en ligne *Oracle Database Administrator's Guide* [12] ainsi que dans le livre *Expert Oracle Database Architecture, Oracle Database 9i, 10g and 11g Programming Techniques and Solutions* [1].

## 3.2 Heap Organized Table (*HOT*)

Il s'agit de l'organisation par défaut offerte par Oracle. Lors de la création d'une table dans ce mode, chaque ligne est stockée de manière aléatoire au sein du ou des fichier(s). De manière imagée, ce type de table peut être comparé à un sac dans lequel des lignes de données sont posées pêle-mêle. Cette structure est souvent combinée avec une autre : l'index. Sans cette combinaison, si le moteur Oracle exécute une requête sur une table *HOT*, il est forcé de lire l'intégralité de la table afin de retrouver l'information. En effet, sans contrainte organisationnelle ou d'unicité, il est impossible de connaître le nombre d'enregistrements au sein de la table *HOT* correspondant aux critères d'une requête. Par conséquent, le moteur d'Oracle est obligé de parcourir chaque enregistrement de la table en le comparant individuellement aux critères de la requête. D'autres structures, telle que la table *Index Organized Table (IOT)* (cf. section 3.4 ), organisent leurs données sur leur clé primaire. Dès lors, lors de l'exécution d'une requête, le moteur Oracle sait que seules les lignes correspondant aux critères définis doivent être parcourues (e.g. accéder aux 10 enregistrements avec les valeurs de clé les plus faibles).

### 3.2.1 Organisation en mémoire

Afin d'illustrer l'organisation d'une table de type *HOT*, l'exemple suivant propose une table contenant des prénoms uniques. Pour rappel, une table stocke ses données au sein d'une structure logique. Au plus haut niveau, une table est stockée dans un segment de type TABLE. Ici, la table PRENOM est contenue dans un segment nommé de la même manière (PRENOM). Un segment ne contient pas d'information, il s'agit d'un conteneur. Au départ, une table ne contient qu'une extension dans laquelle se trouvent ses métadonnées. Par la suite, lors d'ajout d'enregistrements dans la table PRENOM et lorsqu'Oracle détermine qu'il n'a plus assez d'espace disponible au sein du segment PRENOM, il alloue une nouvelle extension composée du nombre de blocs nécessaires. A chaque insertion d'une ligne, Oracle vérifie l'espace existant puis place la ligne dans un espace disponible s'il en existe un, sinon, il alloue une nouvelle extension puis, seulement, place la nouvelle ligne. Celle-ci doit être placée dans une série de blocs contigus. Par conséquent, si une ligne de 6 Ko

doit être placée et que seuls 3 Ko restent disponibles dans deux blocs au sein du segment, Oracle n'utilise pas cet espace, et alloue alors une nouvelle extension afin de placer la dernière ligne [4].

La Figure 3-1 représente l'exemple ci-dessus avec des valeurs concrètes. La première extension composant le segment est remplie avec quatre blocs contenant chacun des prénoms. Chaque bloc peut recevoir des informations jusqu'à 80% de sa taille. Les 20% restants sont réservés pour les mises à jour des données. Après avoir ajouté seize prénoms, le segment PRENOM ne dispose plus d'espace disponible. Oracle fournit alors une nouvelle extension de quatre blocs dans laquelle dix nouveaux prénoms peuvent être ajoutés. A chaque ajout, Oracle cherche un emplacement disponible au sein des deux extensions liées au segment PRENOM. Il est intéressant de constater que chaque bloc ne contient pas le même nombre de lignes car la taille des lignes n'est pas fixe. Les enregistrements sont empilés dans le bloc.

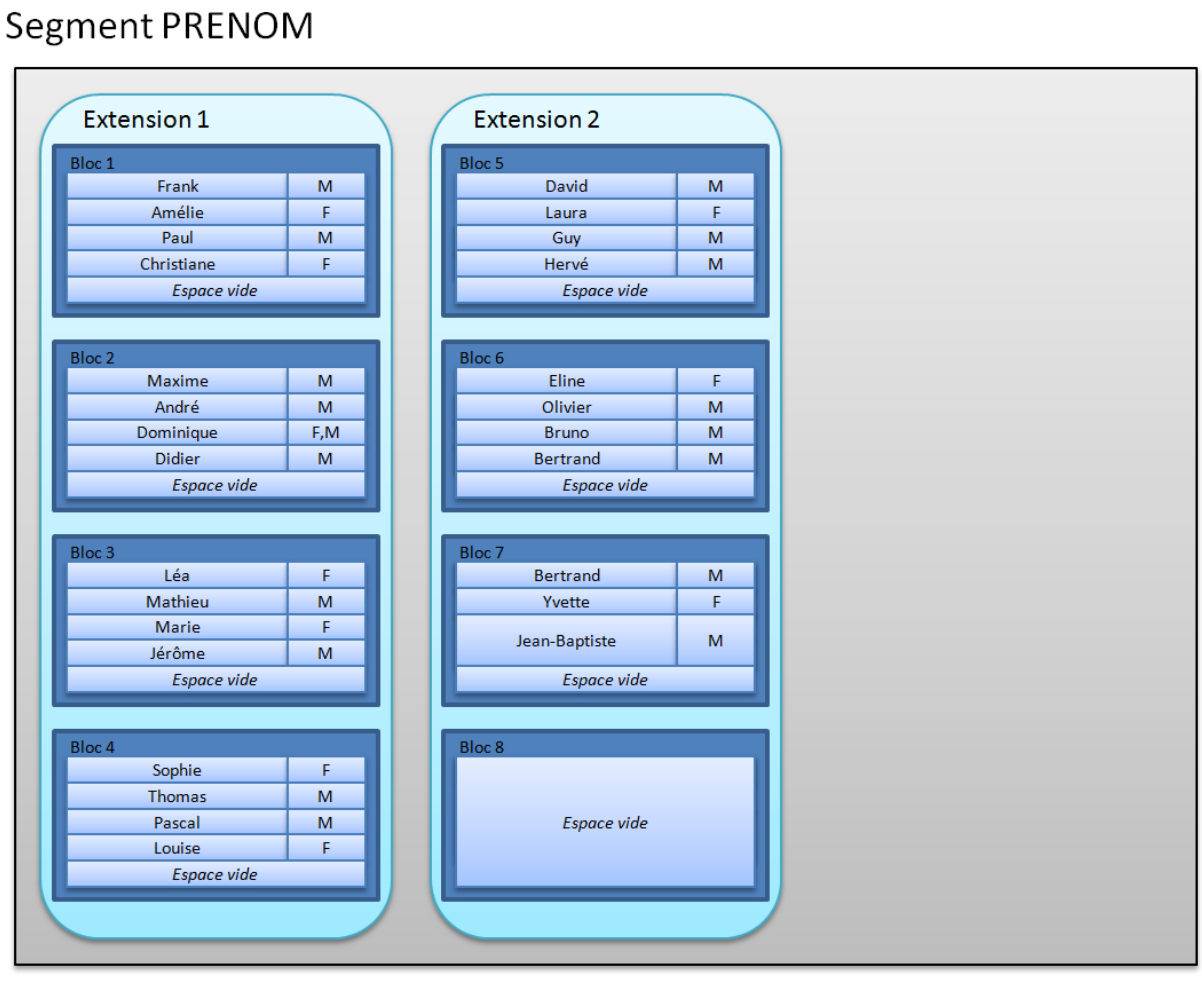


**Figure 3-1 : Représentation de la table PRENOM**

Au final, le segment est composé d'un ensemble d'extensions qui, elles-mêmes, contiennent des blocs qui stockent l'information. Pour connaître les prénoms commençant par la lettre « A », Oracle doit chercher, dans ses dictionnaires, les blocs liés au segment PRENOM, puis charger en mémoire les huit blocs qui constituent la table afin de retrouver les lignes qui correspondent aux critères de la recherche.

### 3.2.2 Les opérations sur une table *HOT*

Le moteur Oracle applique, en fonction de cas spécifiques, différentes opérations sur les tables. Chaque type d'opérations, décrit ci-dessous, est illustré par la table PRENOM qui continue d'évoluer au fil des exemples.

**Insertion au sein d'une table *HOT***

L'insertion est une opération de base qui consiste à ajouter une nouvelle ligne au sein de la table. Dans l'exemple présent, les deux valeurs sont ajoutées : « Jean-Sébastien » et « Jean-Marie-Ange ».

- Ajout de la valeur « Jean-Sébastien »

Lors de l'ajout de « Jean-Sébastien », le moteur Oracle cherche s'il existe une extension qui dispose encore d'espace disponible (N.B. il s'agit d'un prénom composé, il est donc nécessaire de disposer d'au moins deux unités d'espace disponibles). Oracle constate que l'extension 2 dispose de cinq unités de place. Le moteur cherche alors si un bloc dispose de deux places disponibles. Oracle décide de faire l'insertion au niveau du bloc 8, puisqu'il s'agit du premier bloc qui dispose d'un espace suffisant pour accueillir cet enregistrement.

La Figure 3-2 représente le nouveau contenu de la table PRENOM une fois l'opération d'insertion validée.

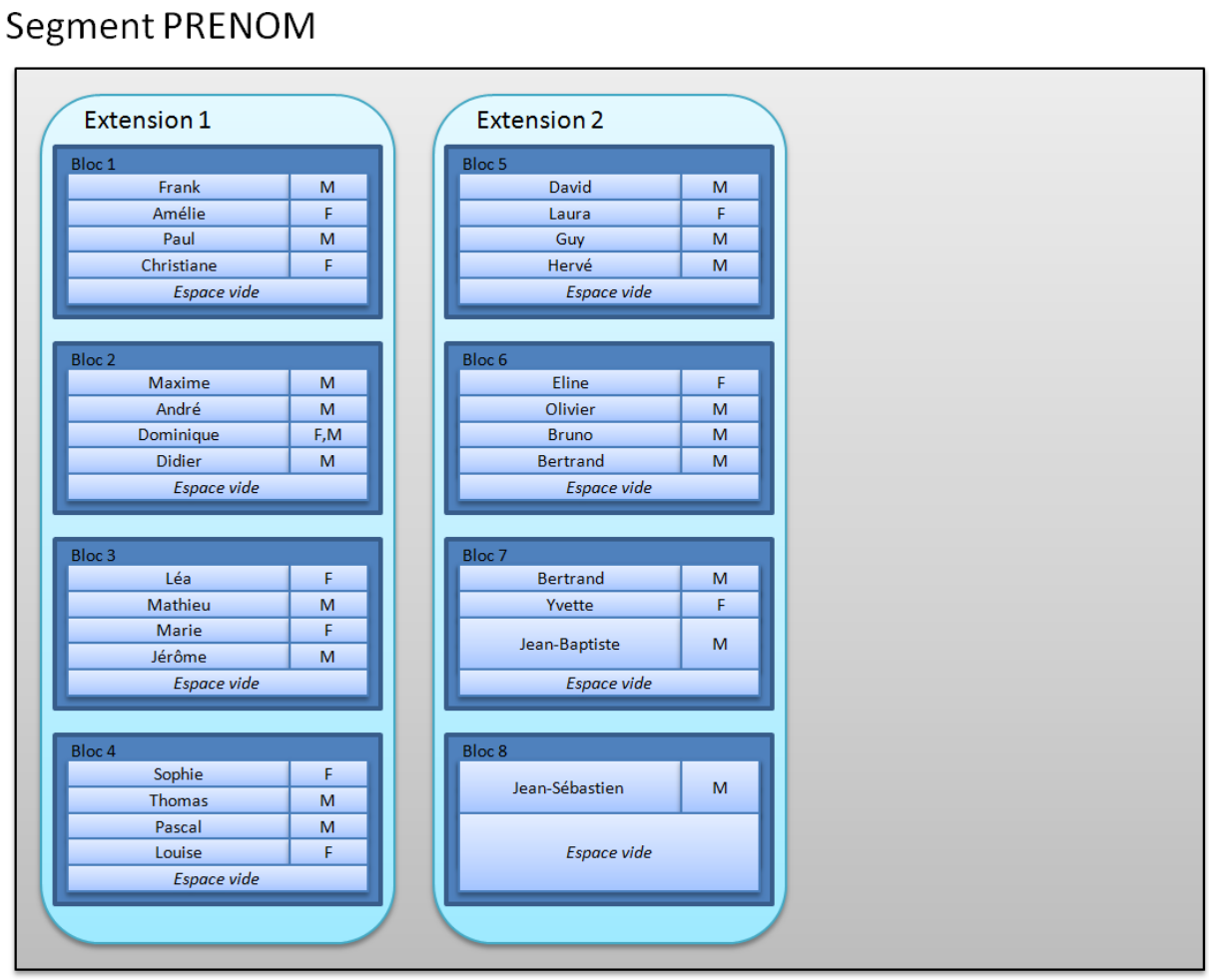


**Figure 3-2 : Insertion de « Jean-Sébastien » dans le Bloc 8**

- Ajout de la valeur « Jean-Marie-Ange »

Pour ajouter « Jean-Marie-Ange », Oracle cherche donc un segment où il est possible de faire l'ajout d'un prénom ayant besoin de trois unités de place pour être stocké. L'extension 2 correspond à cette requête. Par conséquent, Oracle cherche un bloc qui a encore trois unités de place disponibles : ici, il n'en reste pas. Le moteur vérifie si l'enregistrement peut entrer dans un bloc. Si ce n'est pas le cas, Oracle fragmente la ligne dans plusieurs blocs, il s'agit du phénomène de chaînage décrit dans la section 4.3.3 . Si c'est possible de placer la ligne dans un bloc, alors Oracle migre la ligne vers un nouveau bloc qui dispose de l'espace nécessaire. Ce phénomène de migration est décrit dans la section 4.3.2 Dans le cas présent, il est possible de stocker cette nouvelle ligne au sein d'un bloc. Dès lors Oracle fournit une nouvelle extension afin de pouvoir stocker le dernier enregistrement inséré.

La Figure 3-3 illustre la situation après l'ajout du second prénom.

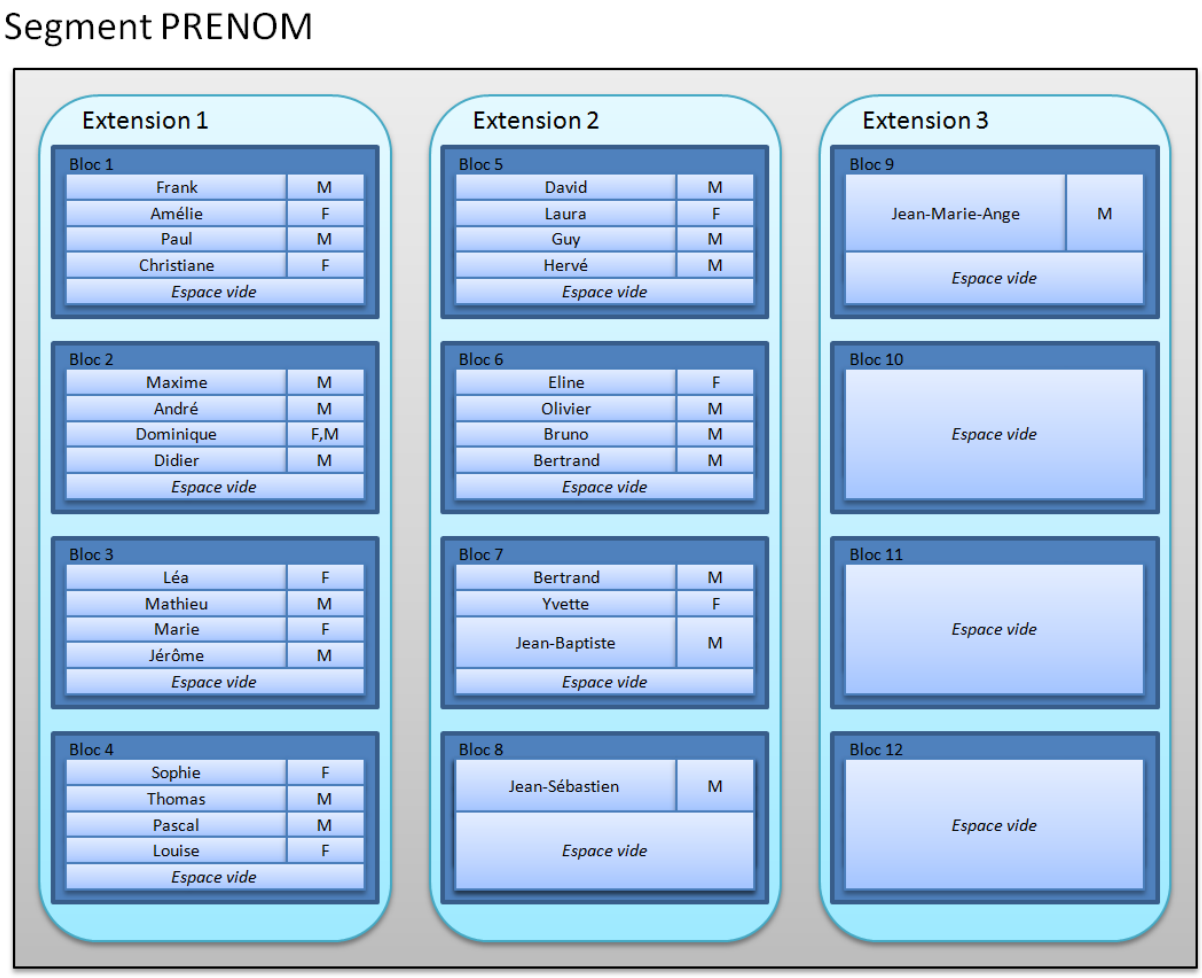


Figure 3-3 : Ajout d'une extension afin d'éviter la fragmentation

**Mise à jour au sein d'une table *HOT***

La mise à jour est une opération qui consiste à modifier une ligne provenant de la table. Deux modifications sont apportées afin d'illustrer ce type d'opération : la valeur « Sophie » devient « Anne-Sophie » et « Louise » devient « Marie-Louise ».

- La valeur « Sophie » devient « Anne-Sophie »

Lors de la première opération, le prénom « Sophie » qui occupait un espace devient composé et doit donc utiliser deux espaces. Oracle vérifie s'il reste de l'espace disponible sur le bloc qui contient

« Sophie » et constate qu'il reste de la place, réservée pour une mise à jour. Cet espace est suffisant afin de faire la mise à jour, par conséquent, celle-ci est effectuée. La Figure 3-4 montre comment l'*espace vide* réservé aux mises à jour est mis à contribution.

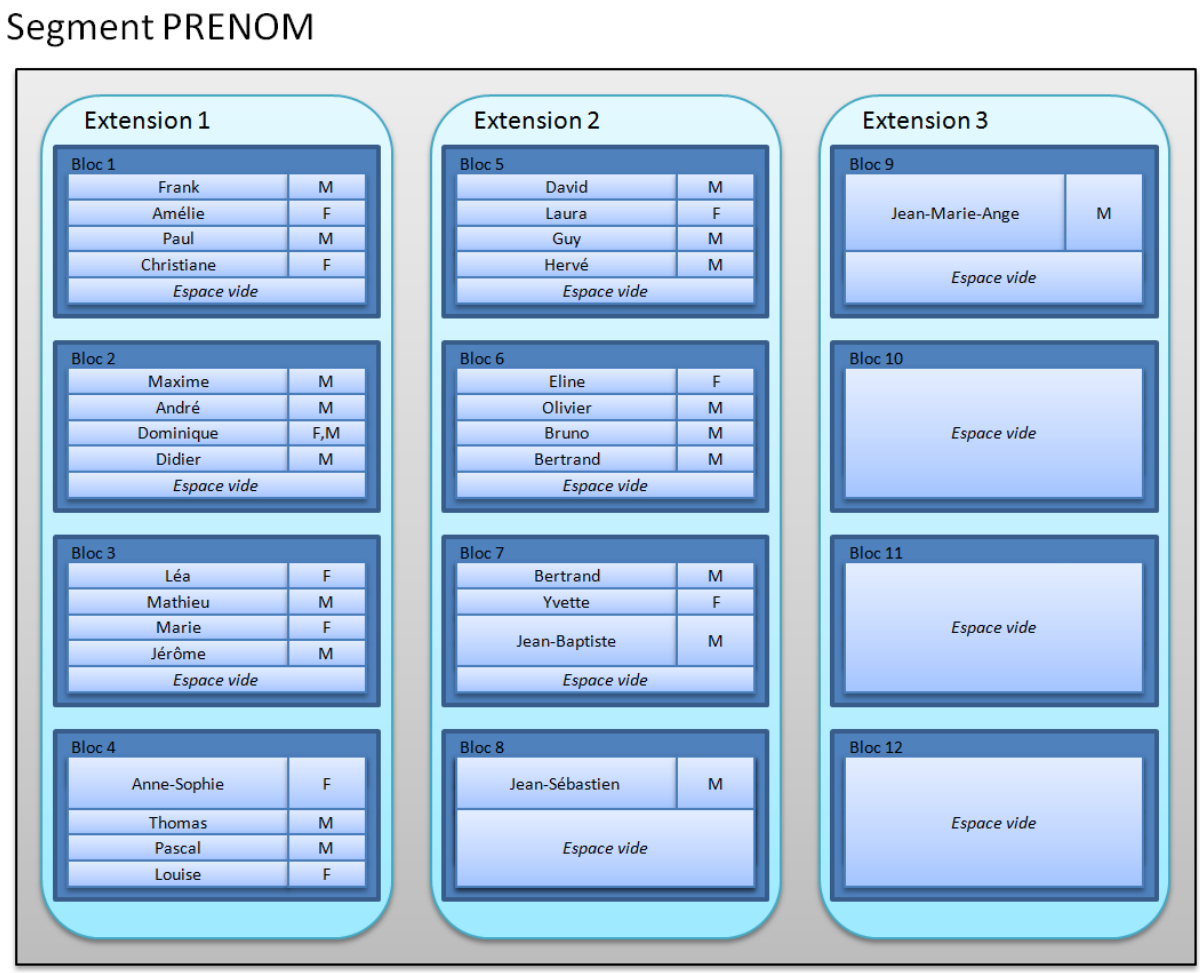

Figure 3-4 : Mise à jour d'une ligne de la table (utilisation de l'espace réservé)

- La valeur « Louise » devient « Marie-Louise »

La seconde opération nécessite une mise à jour du prénom « Louise ». Encore une fois, initialement, le prénom occupait un espace, et sa mise à jour requiert deux espaces disponibles. Cette fois, il n'est pas possible de réaliser la mise à jour au sein du bloc car le bloc 4 est rempli. La ligne subit dès lors une migration. Oracle cherche un bloc qui peut accueillir le prénom mis à jour, puis l'ajoute dans ce nouveau bloc, pour supprimer enfin le prénom d'origine. Dans notre exemple, le bloc 8 accueille le nouveau prénom. Un pointeur est mis en place au sein du bloc 4, signalant ainsi la nouvelle position de l'enregistrement (cf. section 4.3.2 ). La Figure 3-5 illustre le phénomène de migration dû au manque d'espace nécessaire pour effectuer la mise à jour.


**Figure 3-5 : Migration d'une ligne ne disposant pas d'assez d'espace de mise à jour**

### Suppression au sein d'une table *HOT*

La dernière opération qui permet de modifier l'état de la table est la suppression. Dans cet exemple, la valeur « Jean-Marie-Ange » est, dans un premier temps, supprimée. Ensuite, c'est le prénom « Marie » qui est effacé.

- Suppression de la valeur « Jean-Marie-Ange »

Oracle cherche le bloc qui contient la ligne, puis marque celle-ci comme supprimée. Dans ce cas, l'extension 3 est virtuellement vide étant donné qu'elle ne contient aucun enregistrement actif. Elle n'a, à cet instant, plus de raison d'être. Malgré cela Oracle ne fait aucun changement au niveau de la structure de la table, ces quatre blocs restent alloués et seront utilisés lors des prochains ajouts. La Figure 3-6 illustre cette suppression.

Segment PRENOM

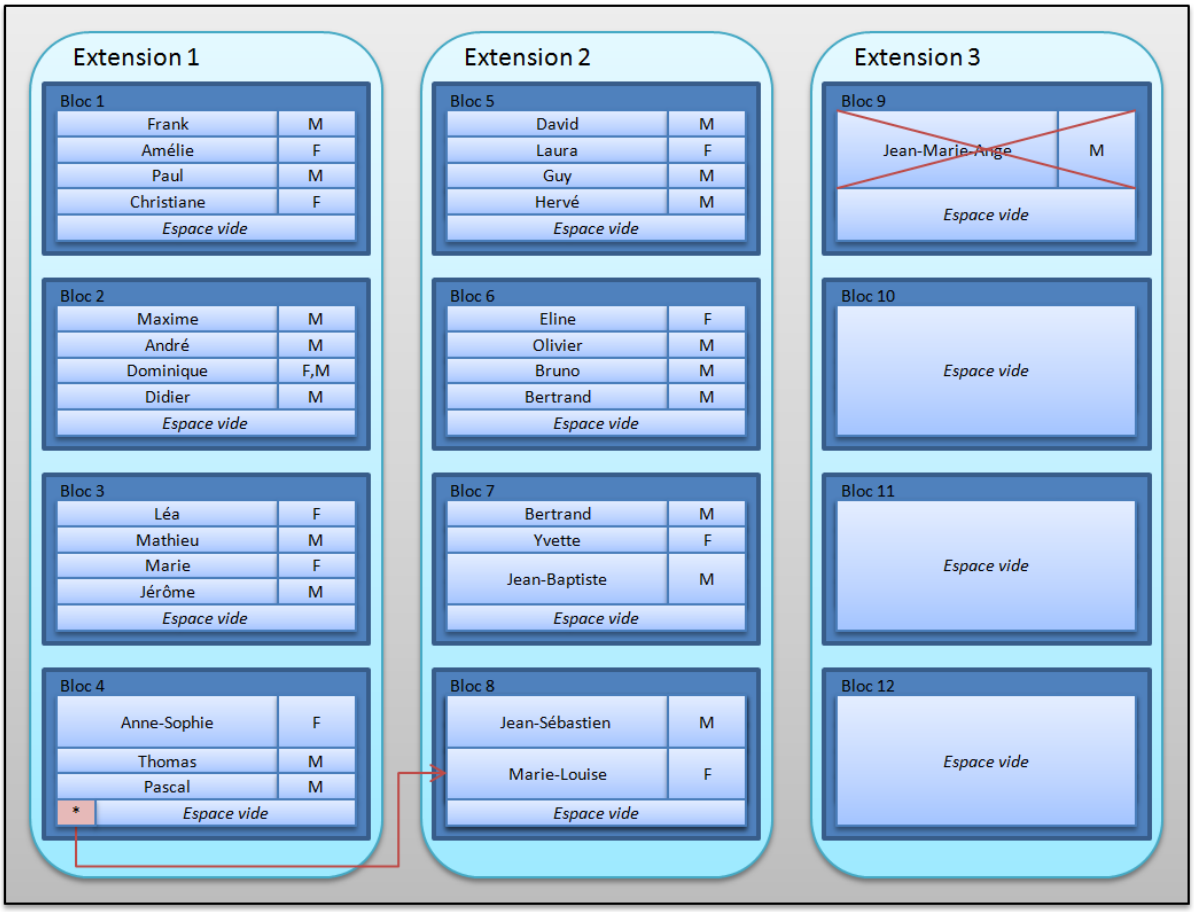


**Figure 3-6 : Après la suppression, une extension est complètement vide**

- Suppression de la valeur « Marie »

La suppression de « Marie » s'organise de manière très similaire à la précédente suppression. Un trou se forme dans le bloc : l'espace vide est fragmenté dans celui-ci. Oracle laisse ce bloc en l'état après la suppression. La Figure 3-7 montre l'espace fragmenté obtenu après la suppression.

Segment PRENOM

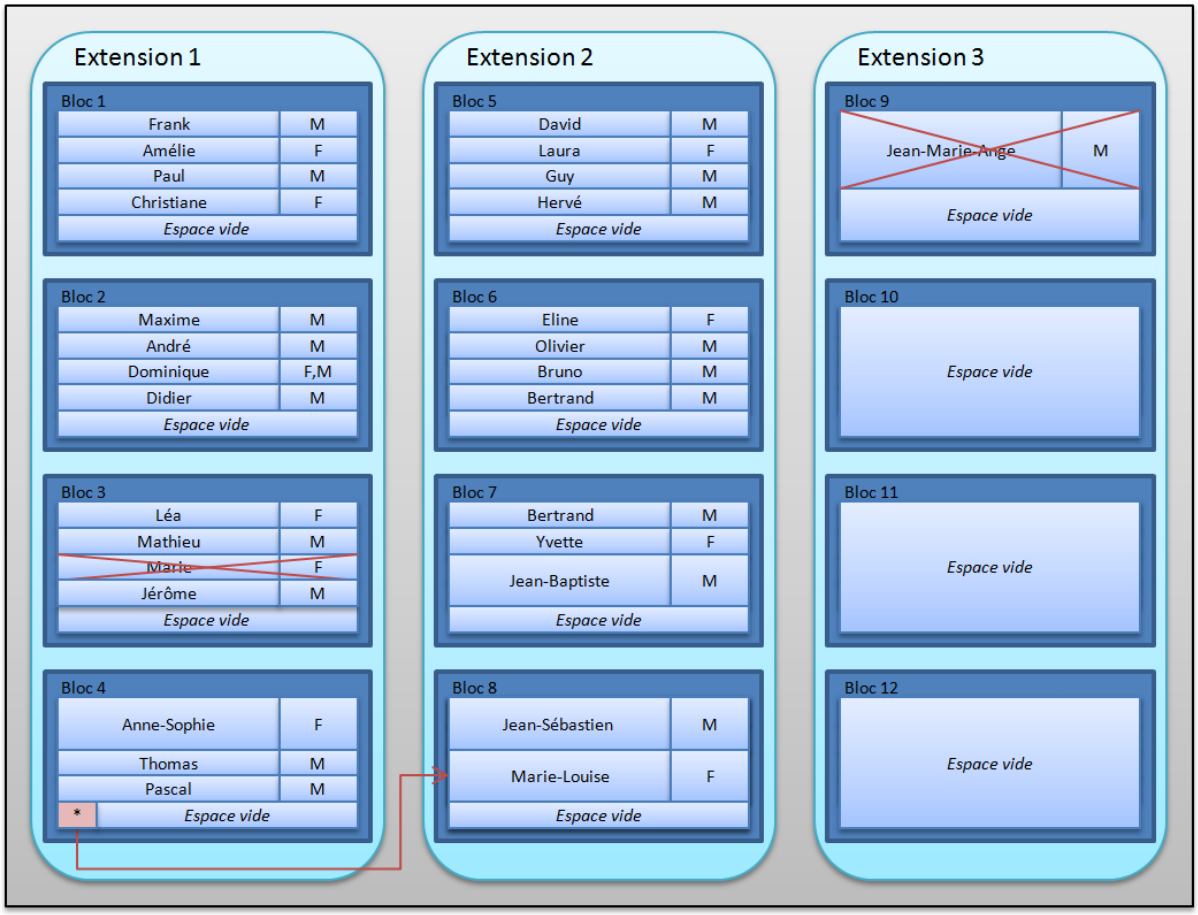


**Figure 3-7 : Espace vide libéré par la suppression de « Marie »**

Si une mise à jour au sein du bloc doit être effectuée, par exemple, « Mathieu » devient « Jean-Mathieu-Philippe ». Oracle détecte que le bloc dispose de l'espace nécessaire mais que celui-ci n'est pas contigu. Oracle se charge alors de réorganiser le bloc de manière transparente afin de ne pas fragmenter « Jean-Mathieu-Philippe ». La Figure 3-8 ci-dessous illustre la façon dont Oracle réorganise le bloc 3 de manière transparente afin d'éviter une fragmentation des informations.

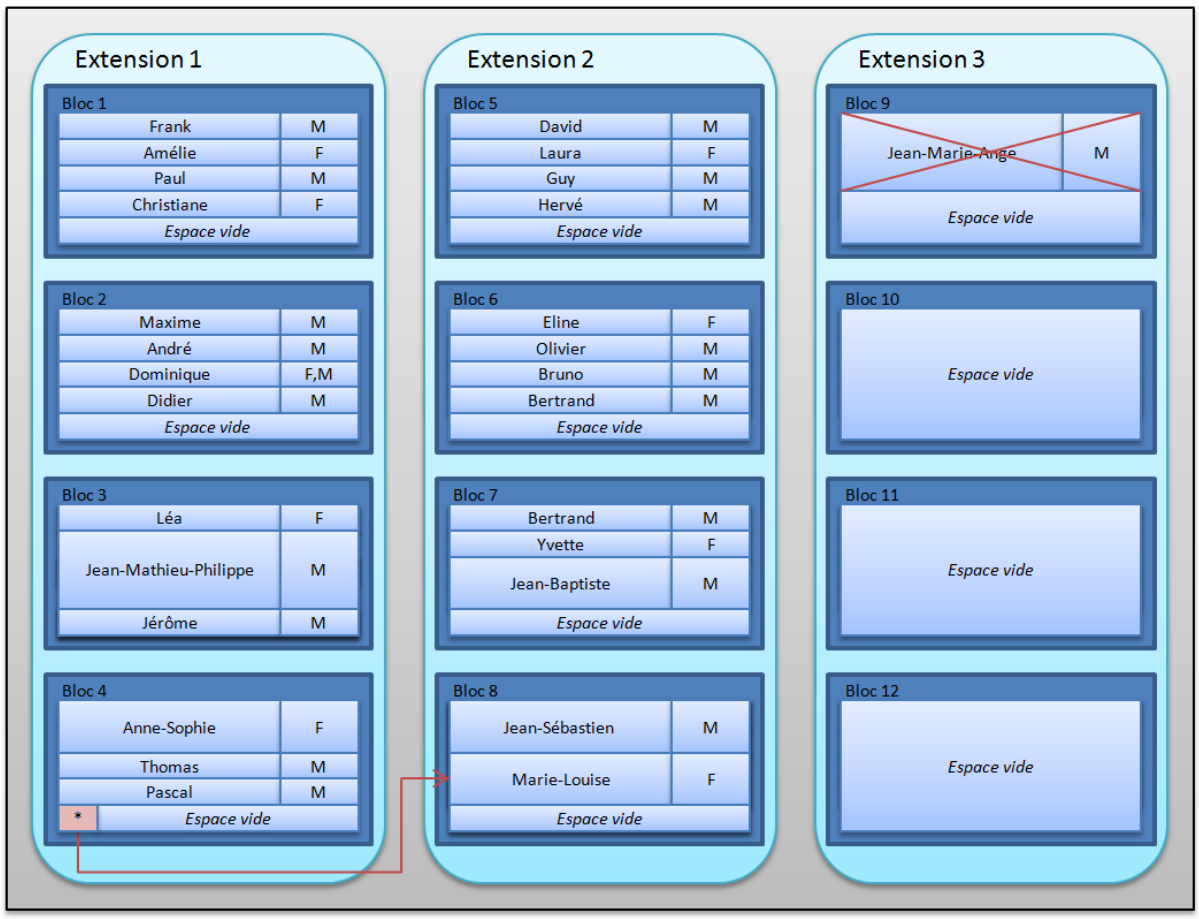


**Figure 3-8 : Réorganisation automatique du bloc 3 après une suppression**

### 3.2.3 Avantages et inconvénients

La table de type *HOT* dispose d'atouts lorsqu'il s'agit d'effectuer des insertions à moindre coût (lecture/écriture et processeur). Son fonctionnement se base sur l'ajout de nouvelles données au premier endroit disponible, ce qui maximise la vitesse d'insertion.

Par contre, cette table n'est pas optimisée pour sa lecture. En effet, étant donné que les informations ont été ajoutées en vrac à l'intérieur de cette dernière, il est obligatoire qu'Oracle parcourt l'ensemble de la table lorsqu'une opération de consultation est réalisée. A mesure que la table grandit, cette opération devient de moins en moins efficiente. Afin de conserver des performances acceptables, il est conseillé de combiner ce type de table avec un ou plusieurs index. Dès lors, le gain de performances obtenu à l'aide de ce type de tables est moins important étant donné qu'il faut maintenir deux objets : une table et son index.

## 3.3 Index *B-tree*

La section précédente donne un aperçu sur la façon de stocker de l'information en vrac au sein d'une table. Heureusement, un *SGBD* offre plus d'avantages qu'un simple espace de stockage. Compte tenu des tailles réellement importantes que peuvent prendre les bases de données, il a fallu trouver un moyen plus efficace que la lecture séquentielle afin de retrouver rapidement l'information. L'index de type arbre (ou *B-tree*) apporte une réponse efficace à cette problématique.

Cet index est la façon la plus commune d'organiser l'information. Il permet de diminuer les ressources nécessaires pour y accéder. L'objectif est d'éviter au moteur de recherche d'avoir à parcourir l'ensemble d'une table pour obtenir l'information demandée par une requête.

### 3.3.1 Organisation en mémoire

L'index de type arbre est constitué de trois types distincts de blocs [7] [12]:

- Le bloc de type « racine » : Il s'agit du premier bloc parcouru. il est toujours placé derrière l'entête du segment. Il s'agit du premier bloc « branche » de la structure [14] ;
- Les blocs de type « branche » : Il s'agit des blocs qui effectuent la jonction entre deux blocs. Ils contiennent une ou plusieurs lignes composées de deux colonnes. Chaque ligne référence un bloc fils de la structure de l'index. Les première et seconde colonnes contiennent respectivement la valeur indexée maximum et l'adresse (de deux octets et demi) du bloc fils correspondant. Ces lignes sont ordonnées ;
- Les blocs de type « feuille » : Il s'agit des derniers blocs de l'index. Ils contiennent une ligne par valeur présente dans la table. Chacune de ces lignes est composée de la valeur indexée ainsi que de l'identifiant unique de la ligne contenue dans la table (*RowID*). Ces lignes sont également ordonnées.

Le *RowID* a une taille de dix octets composé de [12]:

- octets de 1 à 4 : data object id
- octets 5 et la moitié de l'octet 6 : numéro du fichier dans le *tablespace*
- moitié de l'octet 6 et les octets 7 et 8 : numéro du bloc dans le fichier
- octets 9 et 10 : row number dans le bloc

La Figure 3-9 illustre un index créé sur les prénoms de la table PRENOM. Il reste peu d'espace disponible au sein de l'index afin de favoriser les cas particuliers dans la suite de l'explication.

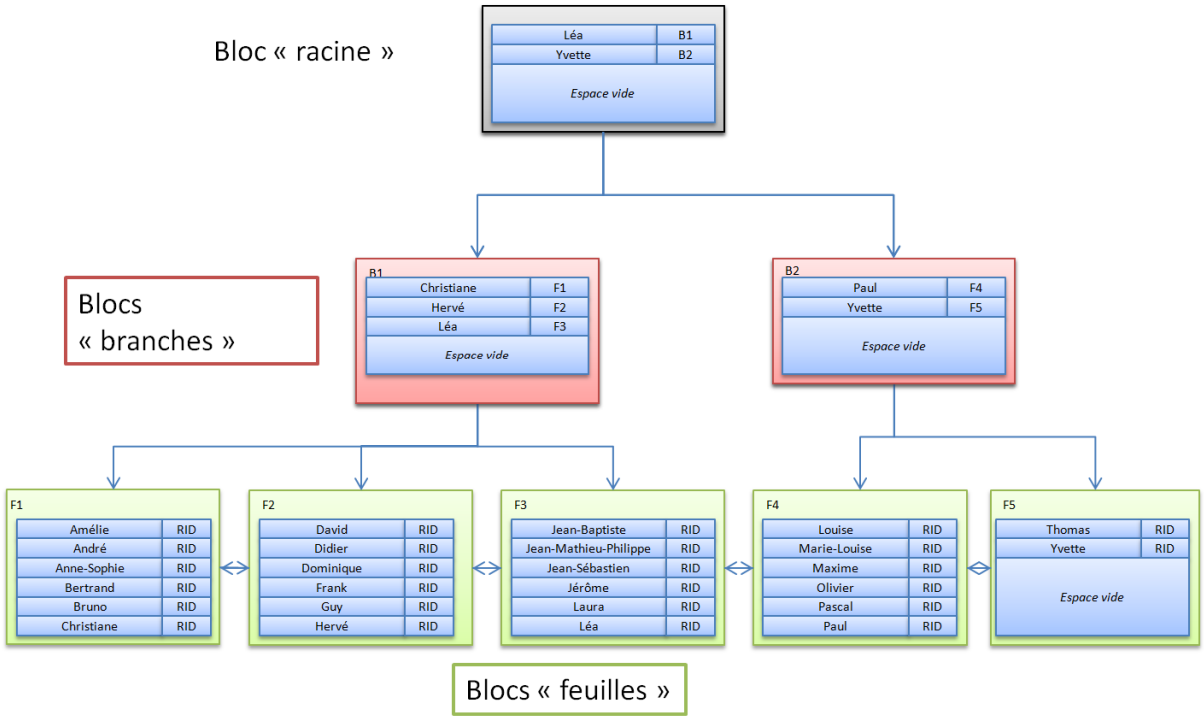


**Figure 3-9 : Illustration d'une construction d'un index *B-tree***

#### 3.3.1.1 La profondeur de l'index

Ce type d'index peut avoir une profondeur (ou hauteur) d'arbre allant jusqu'à une valeur de 4 unités. Le nombre de valeurs stockées dans cet arbre dépend de sa hauteur, de la taille des blocs et de la taille de la valeur indexée en suivant la formule suivante :

$$(\text{Taille bloc} / \text{Taille clé})^{\text{Hauteur de arbre}} = \text{Nombre de valeur maximum}$$

Par exemple, si les blocs pèsent 8 Ko, avec une valeur indexée d'une taille moyenne de 16 octets (6 octets pour la clé primaire et 10 octets pour le *RowID* correspondant) :

- Avec une hauteur d'index de 3 ;

$$(8192 / 16)^{3} = 134.217.728 \text{ valeurs stockables au sein de l'index}$$

- Avec une hauteur d'index de 4.

$$(8192 / 16)^{4} = 68.719.476.736 \text{ valeurs stockables au sein de l'index}$$

La majorité des index Oracle n'atteignent pas une hauteur de 4.

#### 3.3.1.2 Le clustering factor

Le *clustering factor* (*CF*) est un indice calculé lors de la mise à jour des statistiques. Il détermine le coût d'accès à la table en passant par l'index. Il correspond au nombre de lectures réalisées afin d'obtenir toutes les lignes lors d'un *full index scan* [2].

Le calcul du *CF* détermine si l'identifiant de la ligne courante (*RowID*) référence le même bloc au sein de la table que la ligne précédente. Si ce n'est pas le cas, le *CF* est incrémenté d'une unité.

Au final, plus le *CF* est proche du nombre de blocs formant la table, plus l'organisation séquentielle de la table et de l'index sont semblables (cf. Figure 3-10) et moins les ressources seront importantes

pour y accéder. Plus le *CF* est proche du nombre de lignes stockées dans la table, moins l'organisation de la table est ordonnée par rapport à l'index (cf. Figure 3-11).

Un rapport entre le *CF* et le nombre de blocs de la table proche de 1 définit une table parfaitement ordonnée sur la valeur de l'index [8], soit :

$$\frac{CF}{Nbt} \approx 1$$

Le rapport entre le *CF* et le nombre d'enregistrements proche de 1 définit une table parfaitement désordonnée par rapport à l'index [8], soit :

$$\frac{CF}{Ne} \approx 1$$

La formule suivante détermine le nombre de blocs de la table référencés par les clés contenues dans un bloc index :

**Nombre de blocs de la table référencés par un bloc index = CF / $N_{bi}$**

L'indice de perte de performances d'une table *HOT* permet de connaître le coefficient de lecture supplémentaire de blocs à réaliser par rapport à la situation idéale. Il se définit tel que :

**Indice de perte de performance =** $\frac{|\,CF - Nbt\,|}{Nbt}$

Cet indice ne prend pas en compte le taux de rupture au sein des blocs de la table.

Il est important de noter que le *CF* peut être inférieur au nombre de blocs total de la table. Ceci se produit à chaque extension : soit une table avec 1000 lignes ordonnées contenues dans 80 blocs. La table est remplie à 100%, il n'y a plus moyen d'ajouter une ligne. Le *CF* est également de 80. Si une ligne est ajoutée, il est nécessaire de réaliser une extension d'une taille, par exemple, de 8 blocs. Dès lors, 1001 lignes sont contenues dans une table de 88 blocs avec un *CF* de 81.

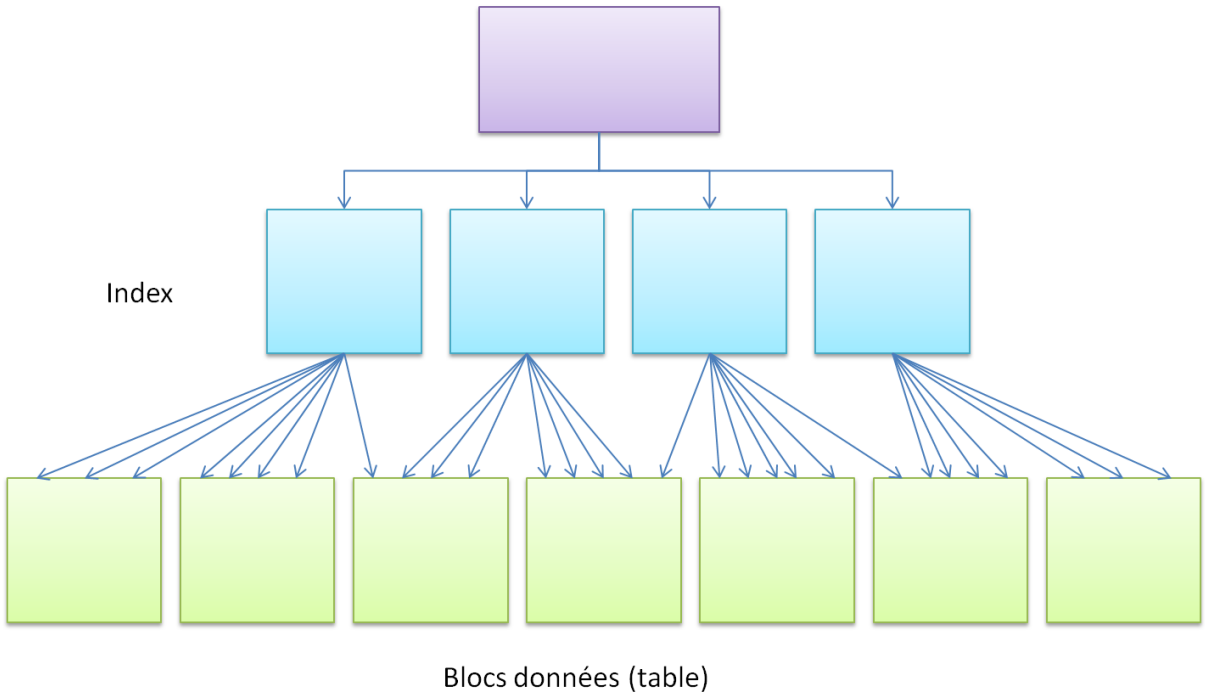


**Figure 3-10 : Illustration d'une table parfaitement organisée sur l'index (Clustering factor faible)**

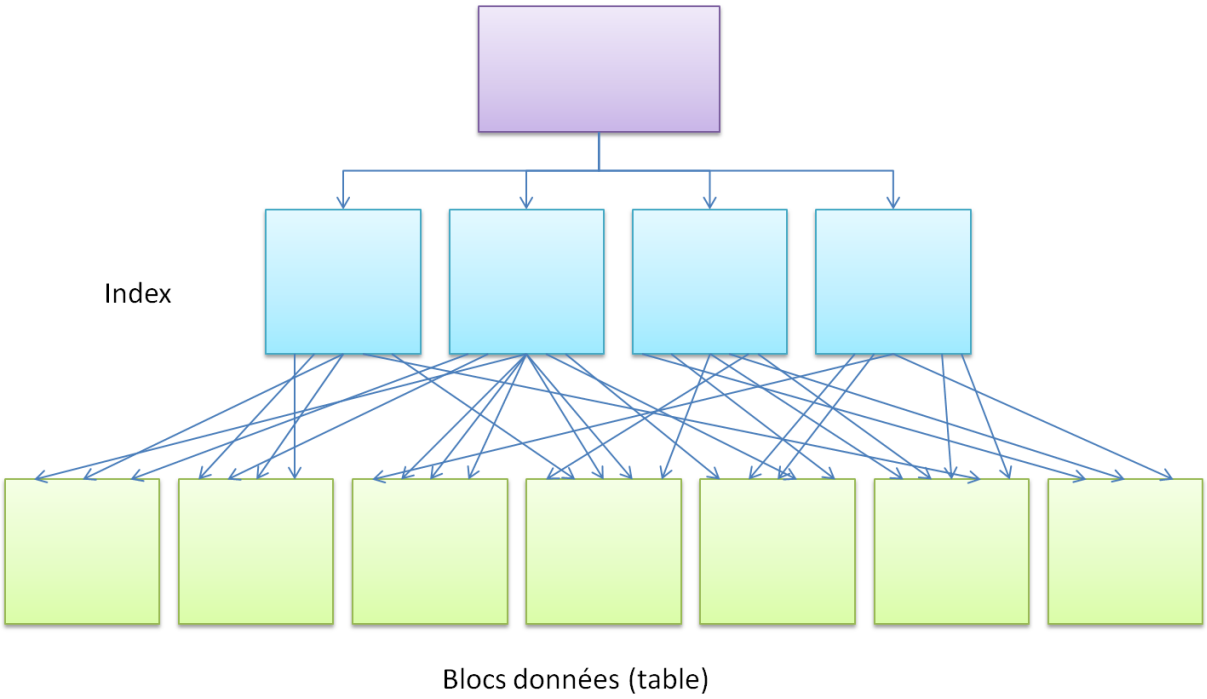


**Figure 3-11 : Illustration une table désorganisée par rapport à l'index (Clustering factor élevé)**

#### *3.3.1.3 Taux initial*

Lors de la création de l'index, le paramètre *PCTFREE* permet de définir le pourcentage d'espace libre disponible laissé à disposition au sein d'un bloc. Cet espace disponible est utilisé afin de faire une insertion sans avoir de phénomène de division. Si cette valeur est placée à 1, chaque insertion oblige la création d'un nouveau bloc, dans le cas de l'autre extrême, si cette valeur correspond à 99, l'espace vide sera trop important, ce qui entraîne deux effets négatifs :

- la taille de l'index est environ 100 fois plus grande que la taille des données présentes lors de la création de l'index ;
- la manipulation de l'index est moins efficiente car elle devra lire plus d'informations.

Par défaut, cette valeur est de 10%. Elle permet de retarder la division du bloc qui est une opération coûteuse.

### 3.3.2 Les Techniques de division sur l'index

Oracle fourni deux méthodes de division lors de l'ajout d'une clé au sein d'un bloc de l'index qui est rempli. La première division est communément nommée 90/10. Elle est réalisée lorsqu'il faut ajouter une ligne au sein du bloc contenant les plus grandes valeurs de clés d'index. Lors de ce type de division, Oracle place la dernière clé ajoutée au sein d'un nouveau bloc qu'il chaîne avec le précédent. Ce dernier est également chaîné avec le nouveau bloc (principe de double chaînage).

Ce type de division est communément nommé 90/10 dans la littérature, et, parfois, est également nommé 99/1 [5]. La notion de 90/10 n'est pas correcte au vu des opérations réalisées dans ce type de division, il s'agit toujours d'une opération de type 99/1. Cependant, Oracle calcule ses statistiques en nommant explicitement « *leaf node 90-10 splits* » pour désigner ce type de division. Par souci de cohérence avec la nomenclature Oracle, le terme 90-10 est utilisé dans la suite de ce document pour désigner ce type de division.

Lors des autres cas d'insertion dans un bloc rempli, Oracle réalise une division de type 50/50. Cette opération consiste à ajouter un nouveau bloc dans la structure de l'index, puis, à répartir la moitié des clés du bloc, celles qui ont les valeurs inférieures, au sein du bloc d'origine, et de placer l'autre moitié des clés à l'intérieur du nouveau bloc.

Chaque fois qu'une division de ce type est réalisée, une rupture de séquence est ajoutée au niveau de l'index. En effet, un nouveau bloc est ajouté au sein de la structure, mais ce bloc se situe sur une extension de plus en plus distante au fur et à mesure que les divisions s'opèrent. Cette notion est désignée par le terme de « rupture ». Les ruptures ne se produisent pas lors de division 90/10.

Les probabilités de ce type d'opération sont calculées dans la section 6.3.4

### 3.3.3 Les opérations sur un index

Lors de la manipulation des données au sein de la table, l'index se voit également modifié. Il existe deux principaux types d'actions : l'ajout et la suppression. La mise à jour correspond à un ajout et à une suppression d'une information au sein de l'index et se décrit par la combinaison de ces actions.

**Insertion au sein d'un index**

Lors d'un ajout d'une ligne au sein de l'index, trois situations se présentent. Dans le cas le plus simple, de l'espace est disponible dans le bloc où doit être insérée la valeur. Celle-ci est dès lors ajoutée au sein du bloc. En ce qui concerne le deuxième et le troisième cas, il n'y pas plus suffisamment d'espace, par conséquent, il faut allouer un nouveau bloc et diviser l'information contenue dans le bloc saturé [5]. Il existe deux types de divisions au sein d'un index Oracle: 50/50 et 90/10.

Les exemples suivant illustrent ces deux techniques.

La Figure 3-12 illustre la situation initiale dont l'index est constitué par les données déjà utilisées précédemment. Bon nombre de blocs sont remplis au maximum de leur capacité afin de favoriser le phénomène de division.

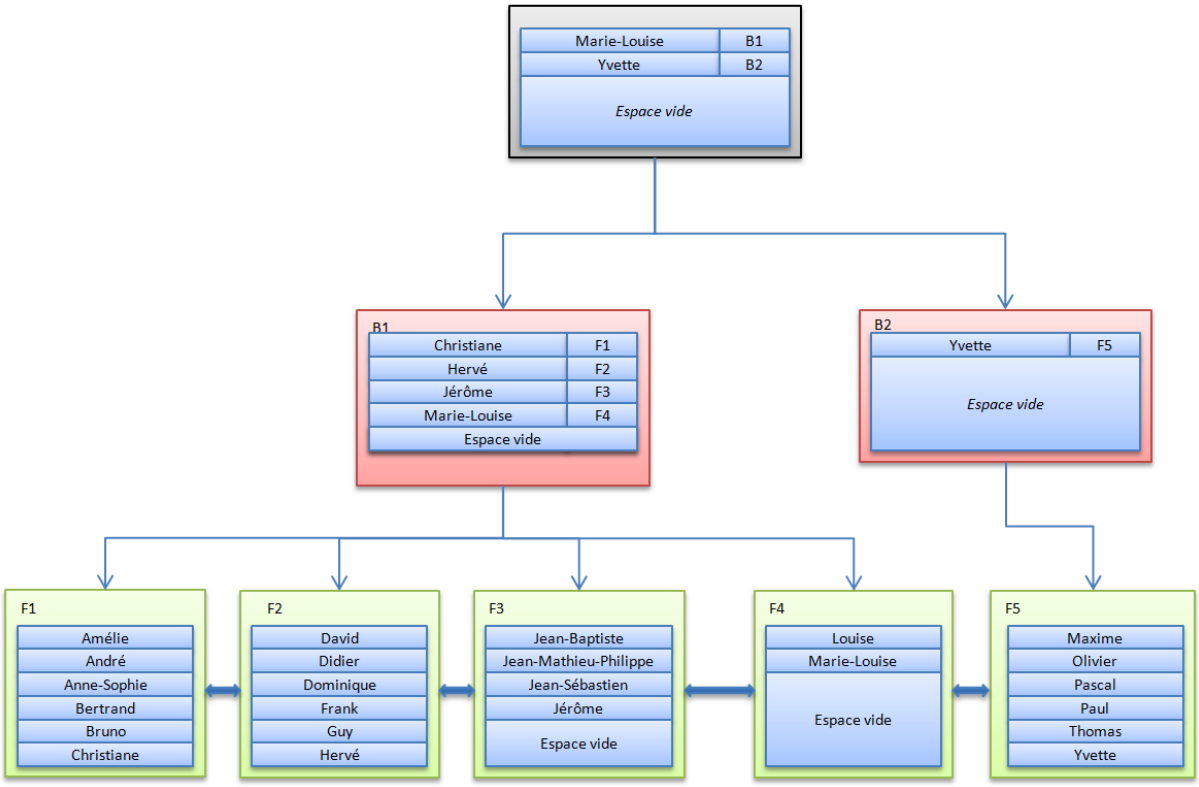


**Figure 3-12: Modification de données - Index initial**

En partant de cet exemple, deux prénoms sont ajoutés : il s'agit de « Léa » et « Laura ». Ces deux prénoms sont ajoutés après « Jérôme » dans le bloc F3. Le bloc F3 est maintenant complètement rempli tel que le montre la Figure 3-13.

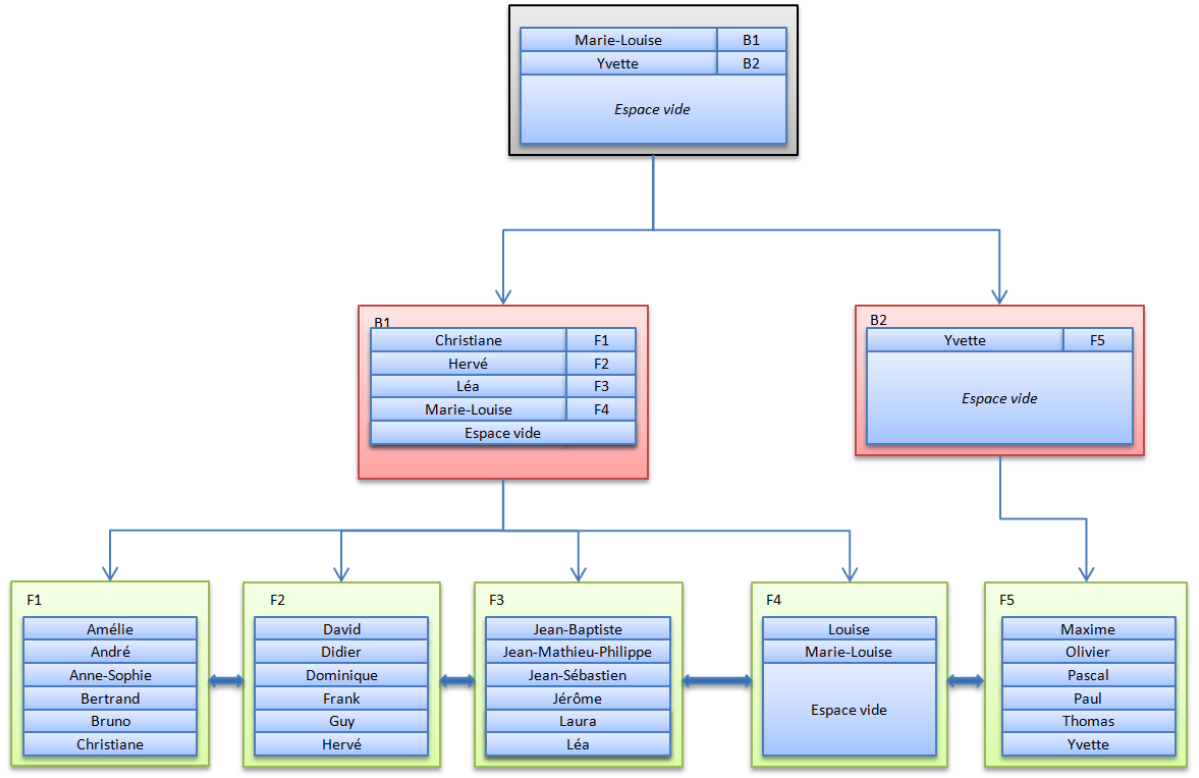


**Figure 3-13 : Ajout de « Laura » et « Léa », le bloc F3 est rempli**

Si, en partant de cette situation, le prénom « Jules » est ajouté au sein de la table, « Jules » doit être ajouté dans l'index entre « Jérôme » et « Laura ». Malheureusement, le bloc F3 est rempli à 100%. Par conséquent, il faut ajouter un nouveau bloc, puis diviser en deux l'information contenue dans le bloc F3. Le nouveau bloc recevra la moitié de l'information. Le bloc « branche » qui référence le bloc « feuille » F3 reçoit la référence du bloc qui vient d'être ajouté. La Figure 3-14 illustre la situation finale après que la division par la technique 50/50 ait été effectuée.

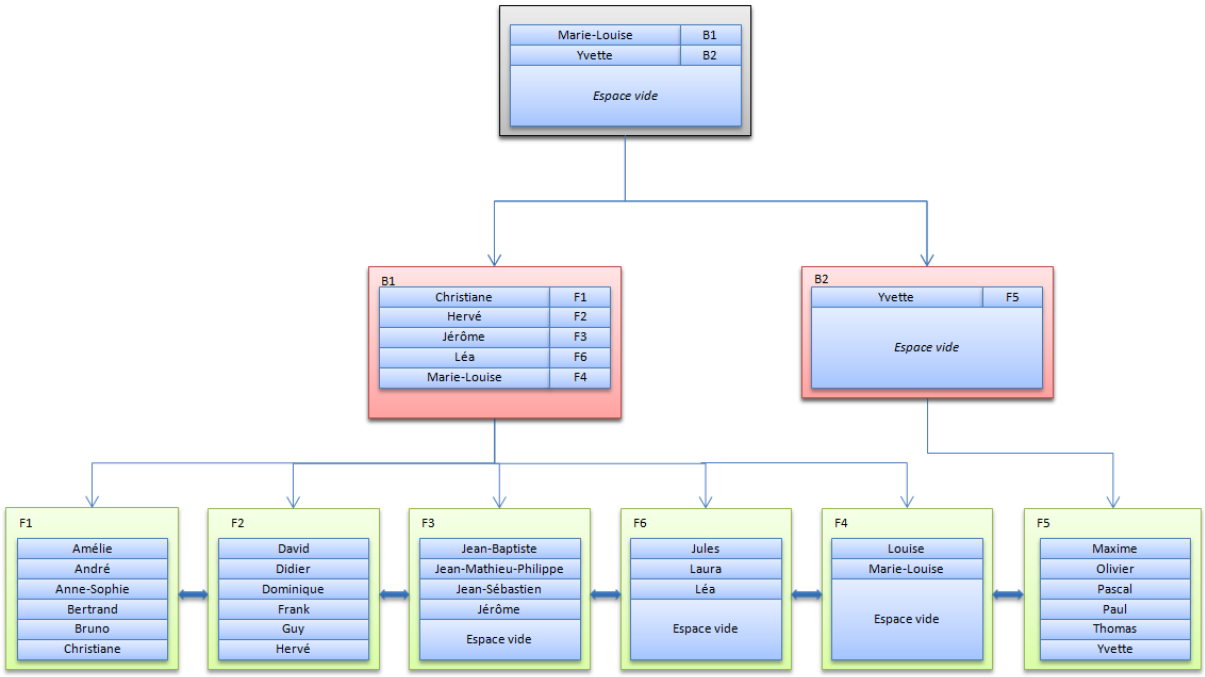


**Figure 3-14 : Modification des données et de leur index lors d'une division 50/50**

Cette opération de division est relativement coûteuse et se passe en six étapes. Il est nécessaire :

- D'allouer un nouveau bloc vide à l'index ;
- De répartir les informations contenues au sein du bloc en deux groupes. Les valeurs inférieures restent dans le bloc initial, les valeurs supérieures sont placées, quant à elles, dans le nouveau bloc ;
- De placer la nouvelle valeur dans le bloc « feuille » correspondant ;
- De mettre à jour le pointeur du bloc initial qui renvoie vers le nouveau bloc ;
- De mettre à jour le pointeur contenu dans le bloc suivant. Ce pointeur doit référencer le nouveau bloc ;
- De mettre à jour le bloc « branche » afin qu'il référence le bloc initial et d'ajouter une ligne spécifiant la valeur inférieure du nouveau bloc.

De plus, le moteur Oracle garde l'index correctement balancé lors de chaque opération d'insertion. Par conséquent, une surcharge significative peut également se produire lorsque qu'un ajout est réalisé et qu'il mène à un déséquilibre du *B-tree* qui compose l'index.

La seconde technique de division dite 90/10 s'applique lors d'un cas particulier. Elle est spécifiquement utilisée dans les cas de figures où des valeurs croissantes sont ajoutées. Lors de l'ajout d'une valeur supérieure à celle du dernier bloc rempli de l'index, Oracle ajoute la nouvelle valeur dans un nouveau bloc. Ainsi, il évite l'opération de répartition de données au sein des deux

blocs, ce qui constitue une opération plus coûteuse. Cette technique est très efficiente pour des tables dont la valeur d'index est strictement croissante (e.g. un *timestamp* ou un *identifiant*) et elle permet de minimiser le surcoût lié à la division d'un bloc.

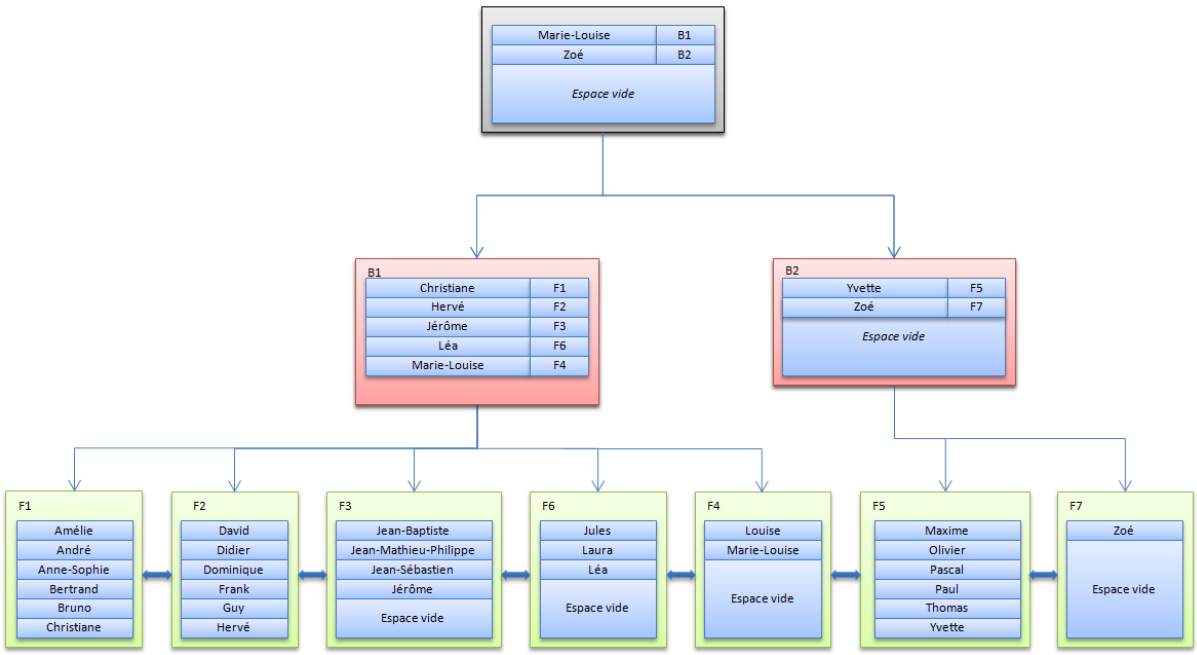


**Figure 3-15 : Modification des données et de leur index lors d'une division 90/10**

Cette opération de division est moins coûteuse que l'opération 50/50. Elle implique seulement :

- D'allouer un nouveau bloc vide à l'index ;
- De placer la nouvelle valeur dans le nouveau bloc « feuille » ;
- De mettre à jour le pointeur contenu dans le bloc initial. Ce pointeur doit référencer le nouveau bloc ;
- De mettre à jour le bloc « branche » correspondant afin d'y ajouter une ligne spécifiant la valeur inférieure du nouveau bloc.

### Suppression au sein de l'index

La suppression dans un index s'organise de manière logique. Lorsqu'Oracle supprime une ligne au sein de l'index, il ajoute un drapeau sur cette ligne signalant que l'espace peut être réutilisé. L'espace n'est pas nettoyé tant qu'une ligne n'est pas ajoutée au sein du bloc. Une fois cette opération réalisée, celui-ci est nettoyé de toute ligne supprimée.

Afin d'illustrer ce fonctionnement, l'entrée « Jérôme » est supprimée. Cette dernière se trouve actuellement dans le bloc F3. Une fois la transaction clôturée, la Figure 3-16 montre que les données concernant ce bloc sont toujours stockées dans celui-ci. Seul un drapeau signale que cet espace est disponible.

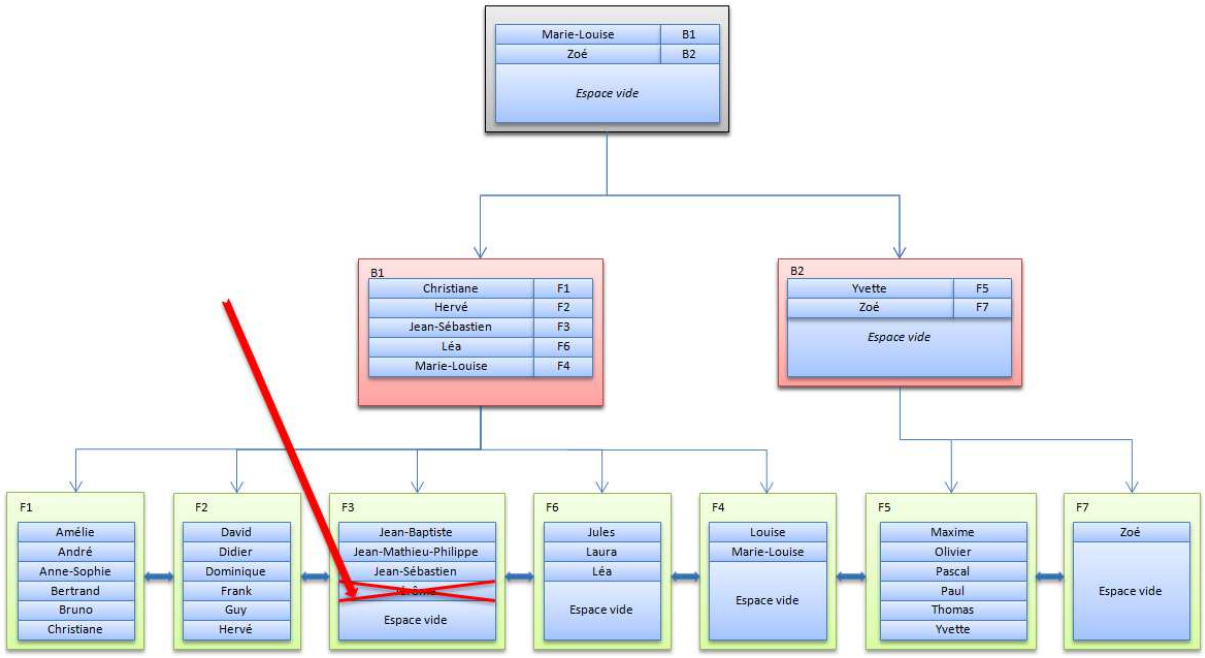


**Figure 3-16 : Suppression de l'entrée « Jérôme »**

Si deux entrées supplémentaires sont supprimées, par exemple « Jean-Mathieu-Philippe » et « Jean-Sébastien », trois lignes au sein du bloc sont notées comme effacées, alors que l'ensemble des informations sont toujours dans ce bloc comme le montre la Figure 3-17.

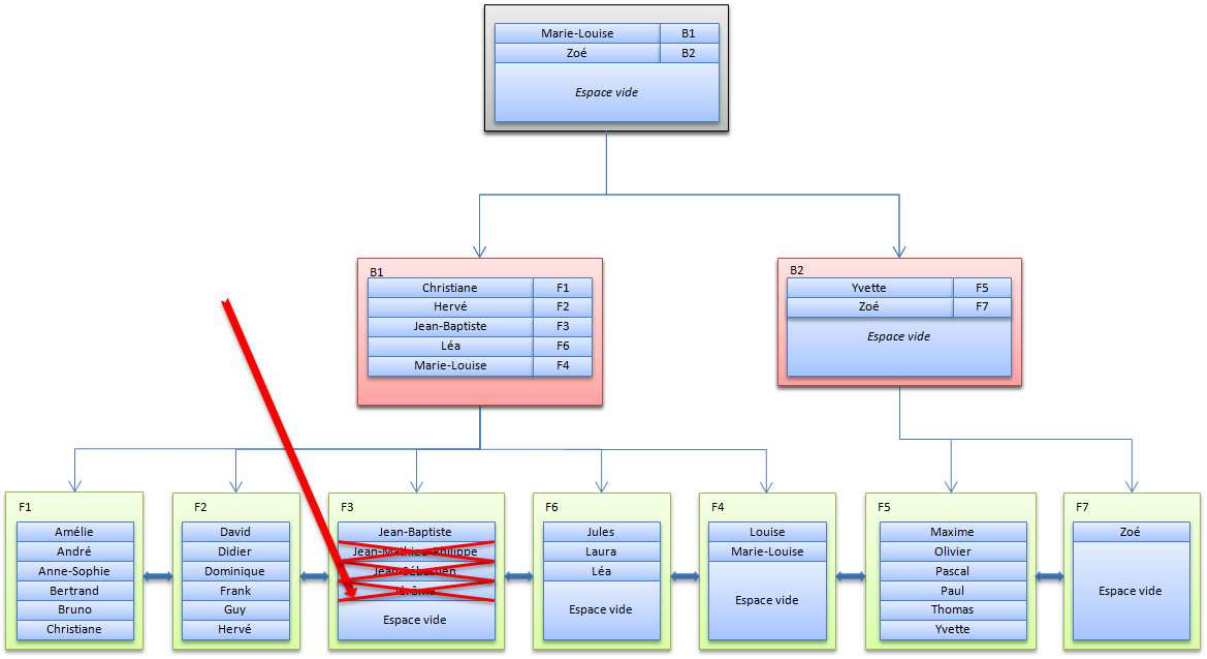


**Figure 3-17 : Etat d'un bloc après de multiples suppressions**

Le recyclage de l'espace est réalisé lors de l'insertion d'une nouvelle valeur au sein de ce bloc. En ajoutant la valeur « Jean », les trois lignes précédemment effacées sont nettoyées et « Jean » est ajouté. La Figure 3-18 présente l'état du bloc après l'insertion.

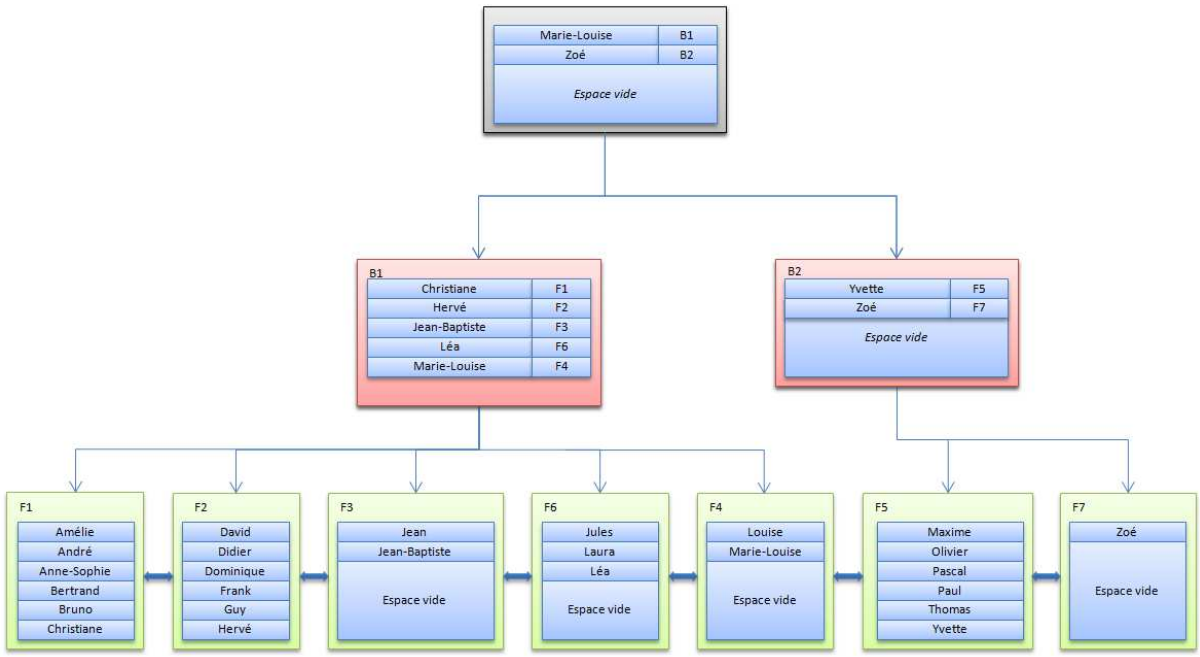


**Figure 3-18 : Nettoyage des suppressions entraîné par l'ajout d'une valeur**

Si l'ensemble du contenu du bloc F4 est effacé, suite à la suppression des valeurs « Louise » et « Marie-Louise », le bloc « branche » qui référençait Louise est aussi affecté par cette suppression comme l'illustre la Figure 3-19.

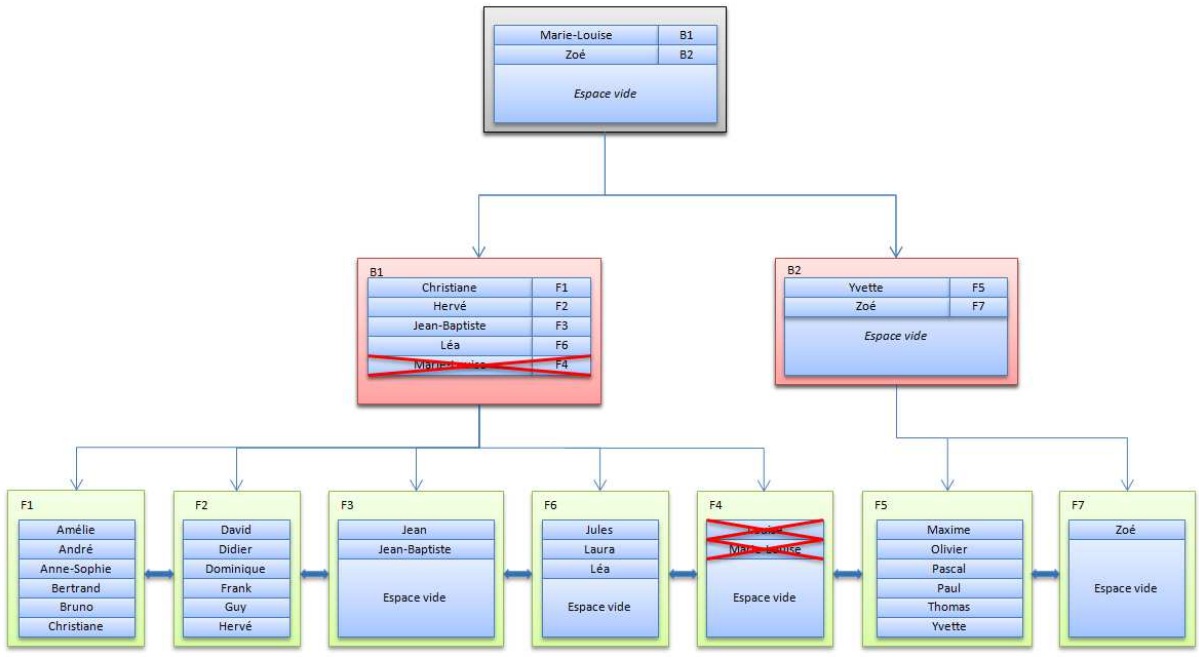


**Figure 3-19 : Suppression de tout le contenu d'un bloc**

Malgré que F4 ne contienne que des données qui sont notées « effacées », le bloc reste dans la structure, disponible pour une future insertion. Lors de l'ajout de « Marie », le bloc est réécrit en ne copiant pas l'ensemble des données qui ne doivent plus apparaitre dans ce bloc. La Figure 3-20: Ajout de « Marie » et recyclage du bloc F4 illustre ce phénomène.

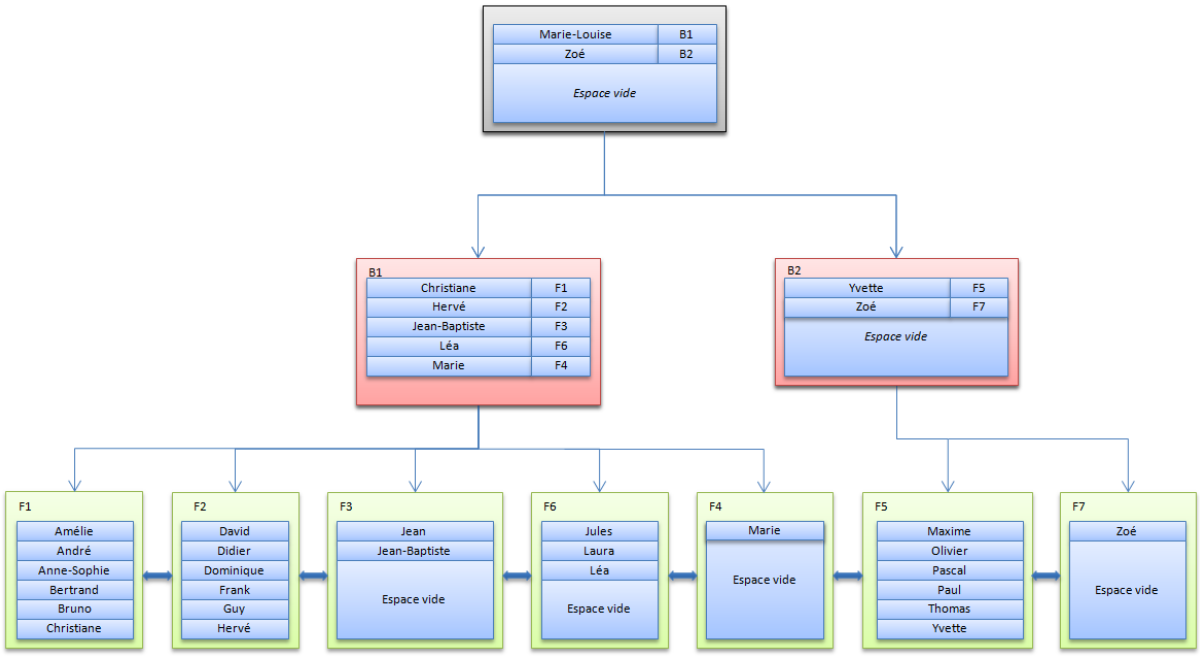


**Figure 3-20: Ajout de « Marie » et recyclage du bloc F4**

### 3.3.4 Avantages et inconvénients

L'index est une structure maintenue dans le but d'augmenter les performances. Il garantit un accès organisé aux données. Il permet d'accélérer grandement la manipulation de tables et évite qu'il faille travailler avec l'ensemble de la table. En contrepartie, il est indispensable de consacrer des ressources pour maintenir cet index lors des opérations d'insertion, de modification ou de suppression d'enregistrements. L'ajout d'index mérite donc d'être réfléchi afin de ne pas générer de surplus d'opérations et ainsi entraver les performances.

## 3.4 Index Organized Table (*IOT*)

L'*index Organized Table* (*IOT*) est une organisation particulière car elle associe deux concepts : Table et Index. Il s'agit d'une table entièrement stockée dans une structure de type index. Dans ce mode, la table est organisée tel un arbre *B-tree*. Lors de la création de la structure, il s'agit de spécifier la clé primaire selon laquelle les données sont organisées. Ensuite cette table se construit à la manière d'un index et sera même enregistrée au sein d'un segment de type « index ». La principale différence avec un index *B-tree* est, qu'à côté de la clé primaire, se trouve l'ensemble de la ligne et non une référence à la ligne [6] [9].

Cette section présente la table *IOT*. Elle n'est pas aussi détaillée que la précédente sur l'index *B-tree* (cf. section 3.3 ) étant donné que les mécanismes régissant leur fonctionnement sont similaires.

### 3.4.1 Organisation en mémoire

Tel l'index *B-tree*, la table *IOT* est constituée de trois types distincts de bloc, dont les lignes sont toutes ordonnées de la même manière que l'index *B-tree* :

- Le bloc de type « racine » : Il s'agit du premier bloc parcouru. il est toujours placé derrière l'entête du segment. Il s'agit du premier bloc « branche » de la structure [14] ;

- Les blocs de type « branche » : Il s'agit des blocs qui effectuent la jonction entre deux blocs. Ils contiennent une ou plusieurs lignes composées de deux colonnes. Chaque ligne référence un bloc fils de la structure de l'index. Les première et seconde colonnes contiennent respectivement la valeur indexée maximum et l'adresse (de quatre octets) du bloc fils correspondant. Ces lignes sont ordonnées ;

- Les blocs de type « feuille » : Il s'agit des derniers blocs de l'index. Ils contiennent une ligne par valeur présente dans la table. Chacune de ces lignes est composée de la valeur indexée ainsi que du reste des valeurs contenues dans la ligne (e.g. colonne 1, colonne 2, colonne 3).

Lors de la création d'une table *IOT* nommée PRENOM_IOT, ordonnée sur le prénom, une structure de type index est obtenue telle que présentée dans la Figure 3-21.

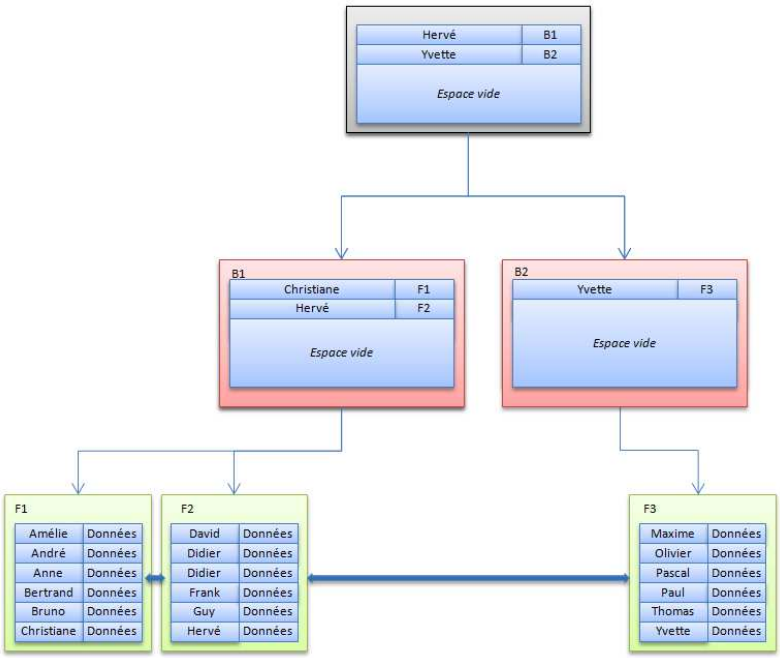


**Figure 3-21 : Organisation physique de la table PRENOM_*IOT***

### 3.4.2 Les techniques de division au sein d'une table *IOT*

Le principe de division des blocs est différent d'un index classique (50/50). Il existe deux types de divisions au sein d'une table de type *IOT*.

Les lignes suivantes définissent ces deux types de divisions et les illustrent avec un exemple pratique. La situation initiale est celle présentée à la Figure 3-21.

Dans l'exemple présent, une table de type *IOT* est définie. Chaque bloc de la table peut contenir 6 enregistrements maximum. La table est ordonnée sur le champ *prénom*.

La première division est de type 90/10. Elle est identique à la division 90/10 décrite à la section 3.3.2 . Elle est réalisée lors de l'ajout d'un enregistrement dont la valeur de la clé est supérieure à toutes celles contenues dans la table *IOT* avant une opération d'insertion. La remarque énoncée dans la section 3.3.2 reste d'application : cette division est en réalité une division de type 99/1. Cependant, Oracle la nomme communément 90/10 dans ses tables de statistiques. Dans un souci de cohérence

avec la littérature et avec la nomenclature Oracle, le terme « division de type 90/10 » est utilisé dans l'ensemble du document.

L'ajout de la ligne « Zoé » provoque cette division puisqu'il s'agit ici d'une division avec une valeur de clé plus grande que celles contenues dans la table. Le nouvel enregistrement est placé dans le bloc F4. La Figure 3-22 illustre cette situation.

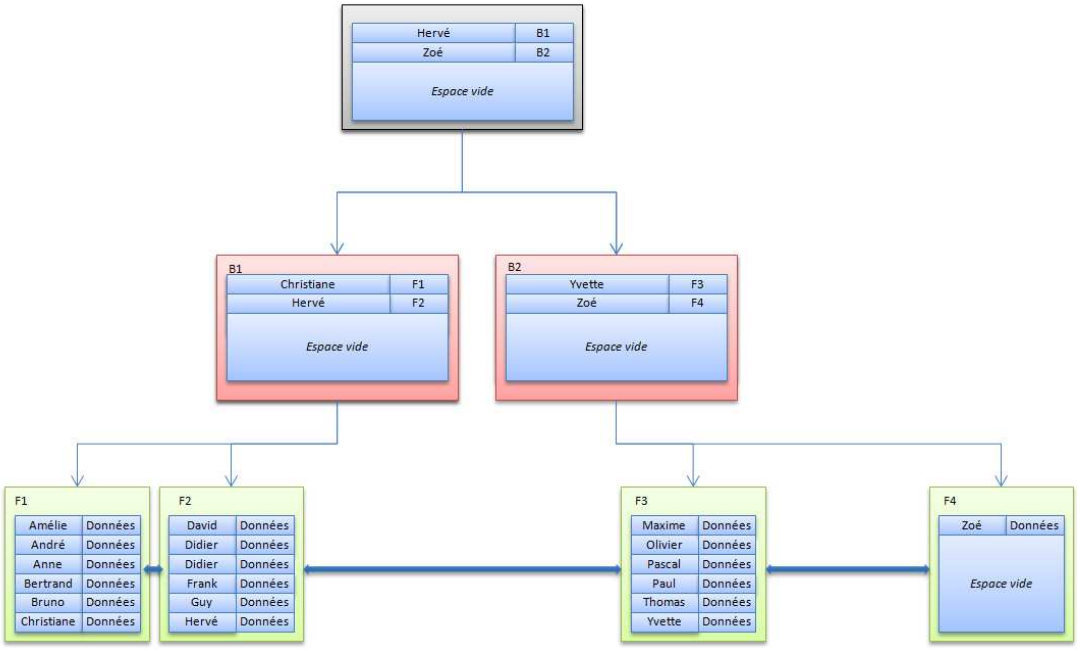


**Figure 3-22 : Ajout de « Zoé » entraînant une division de type 90/10**

Le second type de division se produit, quant à lui, lors d'une tentative d'ajout d'un enregistrement au sein d'un bloc déjà rempli. Dans ce cas de figure, Oracle divise le bloc en deux blocs distincts. Cependant, cette division ne se base pas sur le même principe que la division 50/50 d'un index. Malgré le fait que plusieurs sources référencent cette division comme une division de type 50/50, il s'agit en réalité d'un abus de langage. En effet, lorsqu'Oracle réalise une opération d'insertion dans le bloc courant, l'ensemble des enregistrements, dont la valeur de clé est supérieure à la dernière valeur ajoutée, est placée dans un nouveau bloc. Si la valeur ajoutée est plus élevée que la plus grande valeur du bloc courant, c'est elle qui est placée dans le nouveau bloc. L'ajout d'une clé entre, d'une part, un bloc plein qui contient des valeurs inférieures et, d'autre part, un bloc dont la première valeur est supérieure à la clé insérée, provoque toujours une division *IOT* où la nouvelle valeur est ajoutée seule au sein d'un nouveau bloc. Cette situation provoque un effondrement du taux de remplissage de la table. Lorsqu'une clé inférieure à la première clé contenue dans le premier bloc de la table, est insérée, elle est ajoutée dans ce bloc.

Par conséquent, ce type de division est rarement équilibré tel que la notion 50/50 le suppose. Etant donné que ce type de division ne porte pas de nomenclature bien définie, les lignes suivantes font référence à ce phénomène sous le nom de « division *IOT* ».

Ensuite, on ajoute un enregistrement supplémentaire dont la clé d'index est « Nicolas ». Le *SGBD* essaie de l'insérer dans le bloc F3 qui est rempli. Par conséquent, une division doit être effectuée afin d'obtenir de l'espace supplémentaire. La première question à se poser est le type de division à

appliquer dans la table *IOT*. C'est-à-dire, pour une division 90/10 : la valeur de clé est-elle supérieure à celles des autres clés de la table ? La réponse étant négative, il s'agit alors de réaliser une division *IOT*. Les valeurs inférieures à la dernière valeur ajoutée (« Nicolas »), sont placées dans le premier bloc (F3) avec la valeur « Nicolas ». Toutes les valeurs supérieures sont placées dans le second bloc (F5) tel que le montre la Figure 3-23.

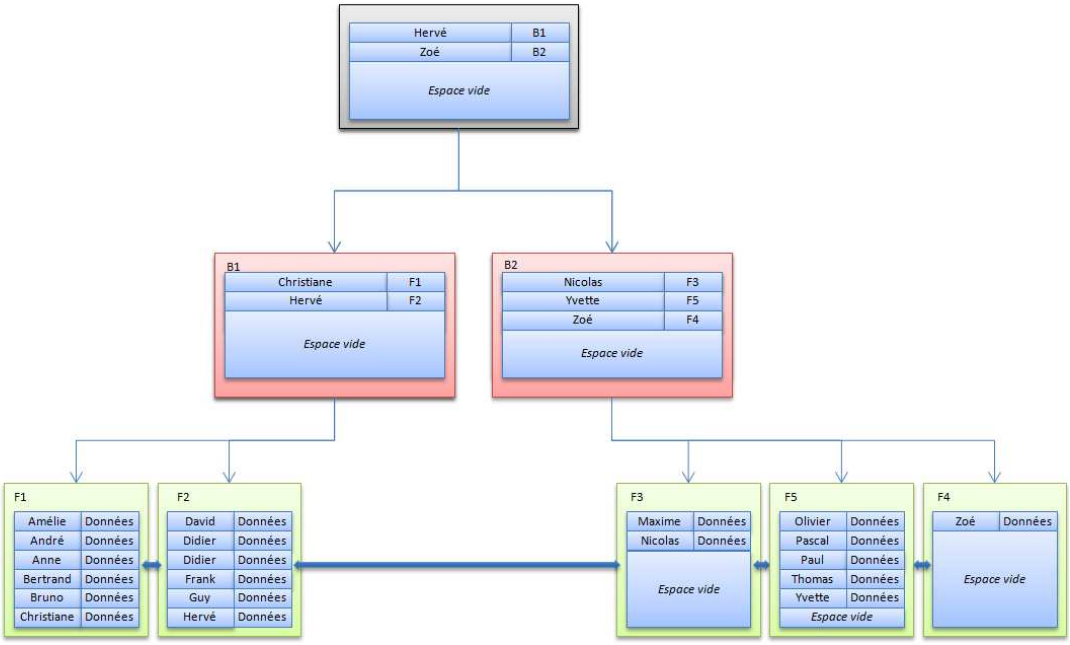


**Figure 3-23 : Ajout de « Nicolas » entraînant une division de type *IOT***

L'ajout d'un enregistrement, ayant pour clé « Manu », doit se faire au sein du bloc F2 car cette clé est également inférieure à « Maxime ». Mais ce bloc contient déjà six enregistrements. Dès lors, une division se produit. La nouvelle valeur de clé est supérieure à toutes les valeurs du bloc F2, il faut donc allouer un nouveau bloc (F6) et mettre l'enregistrement avec la valeur de clé « Manu » au sein de celui-ci. Cette division laisse les six enregistrements initiaux à leur emplacement et place uniquement le nouvel enregistrement au sein du nouveau bloc tel que le montre la Figure 3-24.

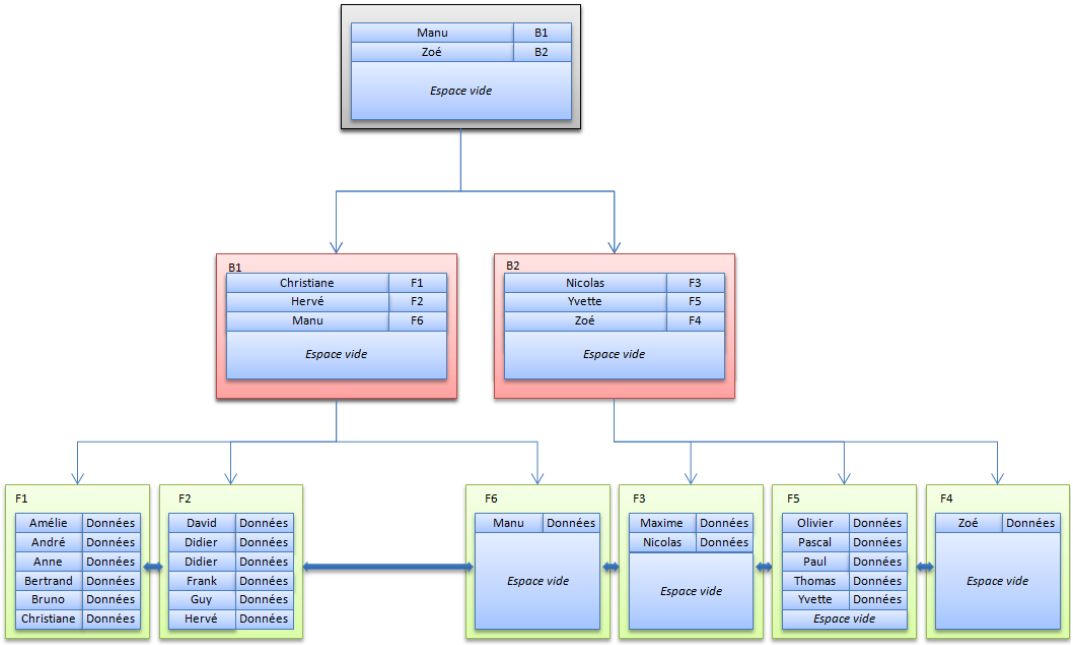


**Figure 3-24 : Ajout de « Manu » entraînant une division de type *IOT***

L'ajout du prochain enregistrement illustre une situation fort défavorable lors de ce type de division. La Figure 3-25 expose l'ajout de « Joëlle ». Il doit se faire au sein du bloc F2 car cette clé est également inférieure à « Manu » (F6). Mais F2 contient déjà six enregistrements. Une division identique au cas précédent, se produit. La nouvelle valeur de clé est supérieure à toutes les valeurs du bloc F2, il faut donc allouer un nouveau bloc (F7) et mettre l'enregistrement avec la valeur de clé « Joëlle » au sein de celui-ci. Cette division laisse, également, les six enregistrements initiaux (F2) à leur emplacement et place uniquement le nouvel enregistrement au sein du nouveau bloc (F7).

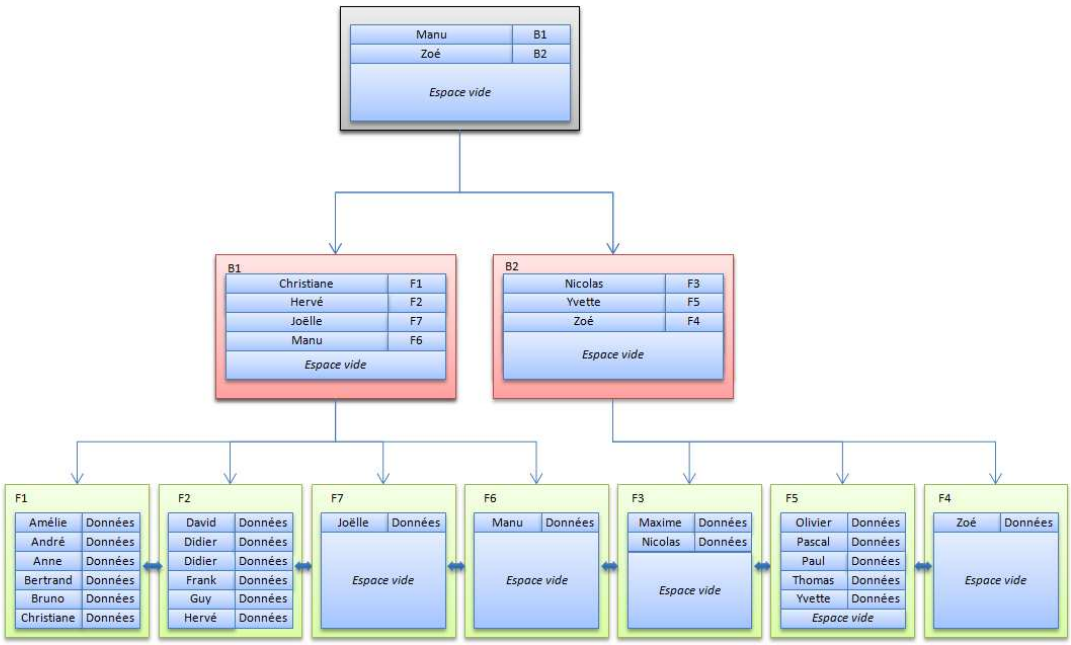


**Figure 3-25 : Ajout de « Joëlle » entraînant une division de type IOT**

Dans le cas de l'ajout de la clé d'index « Amanda ». Il s'agit de la plus petite valeur de la table. Cette clé doit être placée dans le bloc F1. Il faut ensuite recopier les valeurs de clés supérieures au sein du

nouveau bloc (F8), soit l'ensemble des six clés se trouvant dans le bloc d'origine. La Figure 3-26 illustre ce cas.

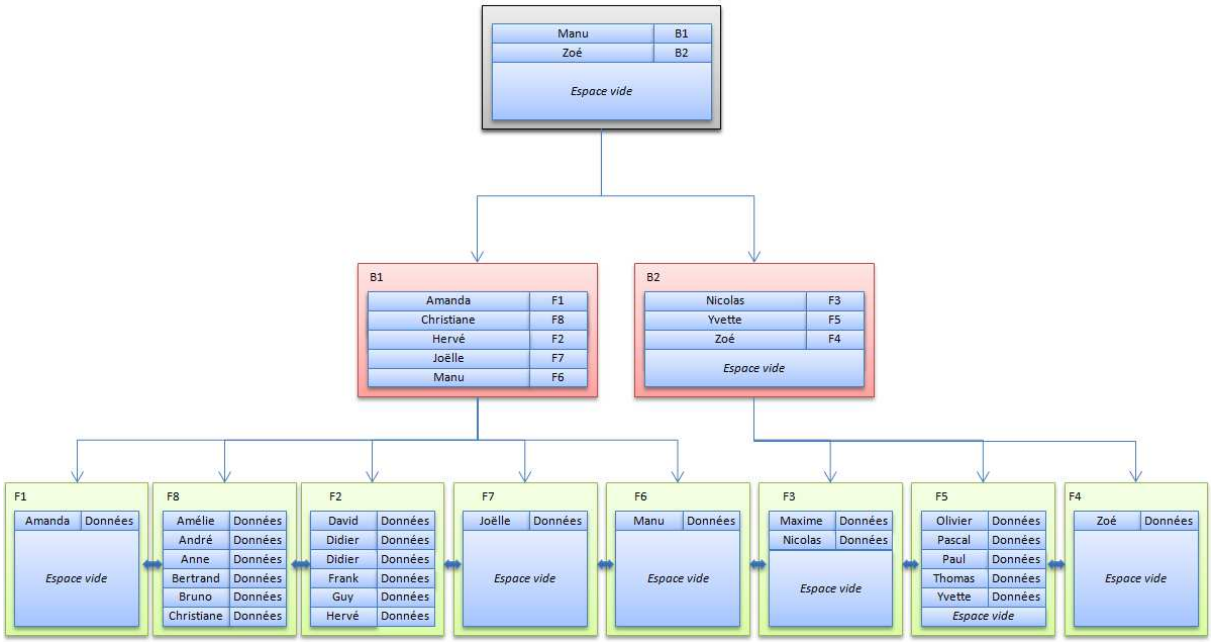


**Figure 3-26 : Ajout de « Amanda » entraînant une division de type IOT**

Enfin, un dernier ajout d'enregistrement est réalisé : l'ajout de la valeur « Adèle ». Il s'agit de la plus petite valeur de la table. Comme précédemment, cette clé doit être placée dans le bloc F1. Ce dernier dispose de suffisamment d'espace, par conséquent, aucune division n'est réalisée. La situation finale est exposée dans la Figure 3-27.

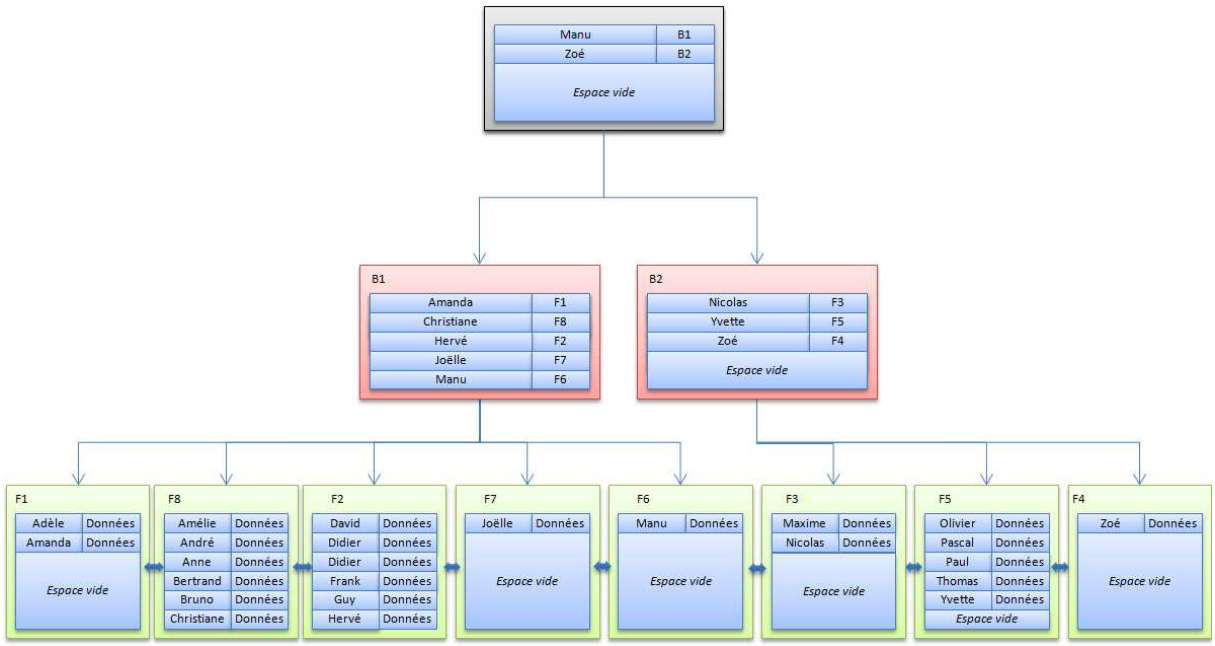


**Figure 3-27 : Ajout de « Adèle » entraînant une division de type IOT**

La situation finale résultant de l'ensemble de ces opérations comporte le bloc « racine » qui référence deux blocs « branche » qui eux-mêmes référencent huit blocs « feuille ». Ces derniers contiennent respectivement deux, six, six, un, un, deux, cinq et un enregistrements. L'équilibre des enregistrements au sein des blocs « feuilles » n'est pas respecté et, surtout, force est de constater qu'il est tout à fait possible d'obtenir un taux d'utilisation inférieur à 50% de la structure *IOT*.

Chaque fois qu'une division de ce type est réalisée, une rupture de séquence est ajoutée au niveau de la table. En effet, un nouveau bloc est ajouté au sein de la structure, mais ce bloc se situe sur une extension de plus en plus distante au fur et à mesure que les divisions s'opèrent. Cette notion est désignée par le terme de « rupture ». Les ruptures ne se produisent pas lors de division 90/10.

Par conséquent, quel que soit le type de divisions au sein de la structure *IOT*, celui-ci est souvent déséquilibré. Ceci amène un comportement significativement différent d'un index classique.

### 3.4.3 Les opérations sur une table *IOT*

Les opérations réalisées sur cette table sont globalement les mêmes que celles réalisées sur une structure index *B-tree*. Par conséquent, cette section présente les similitudes et les différences par rapport à une structure de type index simple.

#### Insertion au sein d'une table *IOT*

L'insertion suit les mêmes mécanismes que l'index *B-tree*. En cas de bloc rempli, Oracle utilise les techniques de la division définie à la section 3.4.1 afin d'allouer un nouveau bloc qui contient les données excédentaires. Ces techniques de division sont excellentes en cas d'insertions de clés croissantes ou décroissantes et maximisent le taux de remplissage de cette structure.

#### Mise à jour au sein d'une table *IOT*

Deux processus de mise à jour distincts doivent être analysés :

- Mise à jour de la clé d'index ;
- Mise à jour des autres données contenues dans la ligne.

Lors d'une mise à jour de la clé d'index, le processus est le même que lors d'une mise à jour d'un index *B-tree*, il s'agit d'une suppression de l'ancienne ligne suivi d'une insertion de la ligne mise à jour.

Lors d'une mise à jour des données qui ne sont pas la clé d'index, Oracle compare la dimension de la ligne mise à jour avec l'espace disponible. Si l'espace disponible est insuffisant, alors il réalise une division du bloc afin de permettre ainsi la mise à jour de la ligne.

#### Suppression au sein d'une table *IOT*

La suppression suit les mêmes processus que ceux de l'index *B-tree*. Oracle commence par placer un drapeau signalant la suppression de la ligne, et lorsqu'au moins une ligne est insérée au sein d'un bloc avec des lignes notées comme « supprimées », alors le *SGBD* réalise le nettoyage du bloc.

#### Lecture au sein d'une table *IOT*

Lors de l'exécution d'une requête, Oracle parcourt la table comme c'est déjà le cas pour un index. Il parcourt la hauteur de l'index afin d'accéder rapidement à l'enregistrement désiré. En cas de lecture

séquentielle, il suffit d'accéder au bloc contenant le premier enregistrement. Ensuite, il est possible de réaliser une lecture séquentielle en parcourant les blocs qui sont doublement chaînés.

### 3.4.4 Avantages et inconvénients

Lors d'une insertion, il est nécessaire de calculer l'emplacement où l'enregistrement doit être ajouté et éventuellement de faire une division d'un bloc afin de pouvoir la stocker. Les mises à jour de clé d'index demandent également un nombre d'opérations (lecture/écriture et processeur) plus élevé que pour une table *HOT* car cette mise à jour requiert une suppression et une insertion. Par contre, le temps d'accès aux données et le scan des valeurs comprises entre deux bornes sont réellement plus rapides.

Il n'est pas très sensé de comparer une table de type *IOT* avec une table de type *HOT* car elles offrent des fonctionnalités fort distinctes. Il semble plus opportun de comparer le fonctionnement d'une table *IOT* avec le fonctionnement d'une table *HOT* qui disposerait d'un index sur sa clé primaire.

A partir de ce postulat, dans certains cas, la table *IOT* offre des performances bien plus élevées. Par contre, elle n'offre pas la souplesse d'une table *HOT* puisqu'il n'est pas possible de séparer l'index de la table. Par conséquent, l'insertion et les mises à jour massives font peser une charge supplémentaire significative qui peut, dans certaines hypothèses, rendre la table *IOT* réellement moins performante.

## 3.5 Synthèse

Cette section vise à présenter trois types d'objets utilisés pour la suite de l'étude. Pour chacune de ces structures, un ensemble d'éléments est décrit : Leur structure, les opérations qu'il est possible d'effectuer, la manière dont chaque opération impacte la structure, les avantages et inconvénients et les principes de divisons.

Le premier type de ces objets, la table *HOT*, constitue un conteneur afin de stocker un ensemble désorganisé d'enregistrements. Dans cette section, les opérations d'insertion, de suppression et de modification sont modélisées pour cet objet afin d'en comprendre le fonctionnement. La table *HOT* est la structure utilisée par défaut au sein d'Oracle.

Le second type objet, l'index, est intimement lié à la table *HOT*. Il constitue un conteneur organisé de type B-Tree. Cette organisation permet un accès plus rapide à une table. Par soucis de cohérence, les opérations sont les mêmes que celles réalisées sur la table *HOT*. Celles-ci sont cependant gérées différemment pour l'index. En effet, en raison de l'entretien de la structure B-Tree, les opérations entraînent un coût en ressources supplémentaires. Un index est toujours lié à un autre objet. Pour la suite de ce document, l'index est lié à une structure *HOT*. Cette section définit son organisation en mémoire, puis donne plusieurs mesures qui lui sont spécifiques. Le *CF* est une notion importante qui mesure de taux d'organisation des lignes contenues dans la table *HOT* par rapport aux lignes de l'index. Ensuite, les techniques de division 90/10 et 50/50 sont expliquées. Enfin, les opérations sur cet objet sont détaillées.

Le troisième et dernier type d'objet est la structure *IOT*. Elle est également un conteneur organisé de données de type B-Tree. L'organisation de cette mémoire est proche de celle utilisée par l'index classique. type de structure met en œuvre une division particulière explicitée par un exemple. Enfin, les avantages et les inconvénients liés à cette structure sont évalués.

L’ensemble de ces structures permet à la fois de stocker les données et d’y accéder. Les mécanismes de B-Tree mis en place permettent un accès rapide à l’information mais, en contrepartie, augmentent les ressources nécessaires lors des opérations d’insertion et de modification.

# 4 Synthèse des mécanismes de fragmentation

## 4.1 Préambule

Précédemment, les différents mécanismes utilisés pour stocker l'information et la restituer ont été abordés. Une des caractéristiques principales d'un *SGBD* est qu'il est appelé à être, en permanence, modifié. Ceci a un impact sur les performances : au fur et à mesure de son utilisation, un *SGBD* perd en effet de son efficience.

D'où provient cette baisse de performances ? Comment se définit la fragmentation et quelles opérations augmentent ce phénomène au sein d'Oracle ?

## 4.2 Principe de base de la fragmentation

Le principe de fragmentation est un élément lié à la manière de stocker l'information. Lorsque l'information est stockée, elle est morcelée en plus petites unités qui peuvent ne pas être organisées en un bloc. La fragmentation est un phénomène défini comme étant le morcellement désordonné, au moins partiel, d'une information qui forme un tout ordonné.

Plus un objet est grand, plus il risque de devoir être fragmenté puisqu'il devient plus difficile de le stocker d'un seul tenant sur un support. Plus un objet subit des modifications, plus il risque de présenter des fragments au sein de l'organisation de ses données.

Lors d'une lecture séquentielle de l'information, chaque fragment entraîne un coût d'accès supplémentaire. Par conséquent, plus un système est affecté par ce phénomène de fragmentation, plus ses performances s'en trouvent affectées.

## 4.3 La fragmentation au sein d'oracle

Comme tout autre système, Oracle est également impacté par la fragmentation. Plusieurs éléments sont impliqués dans les mécanismes de dégradation de performance et entraînent une fragmentation de l'information au sein du *SGBD*. Ces éléments sont :

- La structure logique des fichiers ;
- Le phénomène de migration ;
- Le phénomène de chaînage ;
- La manière d'insérer les données.

### 4.3.1 La structure logique

Tel que décrit dans la section 2 , Oracle organise ses données au sein d'une structure logique. Les données sont contenues dans un segment lui-même constitué d'une succession d'extensions. Le moteur Oracle part du principe que chaque extension est une suite de blocs contigus. Chaque extension est donc considérée comme étant d'un seul tenant. En pratique, les extensions sont des fragments d'un même segment, rarement contigües. Par conséquent, chaque extension constitue un fragment supplémentaire de l'information.

### 4.3.2 Migration

Lorsqu'une ligne subit une mise à jour qui induit une augmentation de sa taille de telle façon qu'elle ne puisse plus être stockée dans le bloc courant, cette ligne est déplacée dans un bloc disposant de suffisamment d'espace. Afin de permettre une redirection vers le nouvel emplacement de

l'enregistrement, un pointeur est défini au sein du bloc. Cette opération est appelée migration [10] [6].

Ci-dessous une illustration simple de ce phénomène.

Situation initiale:

Le *SGBD* est configuré pour fonctionner avec des blocs de 8 Ko

```
CREATE TABLE migration(
 id int PRIMARY KEY,
 data1 CHAR(2000) ,
 data2 CHAR(2000) ,
 data3 CHAR(2000) ,
 data4 CHAR(2000) ,
 data5 CHAR(2000) ,
 data6 CHAR(2000)
);
```

Lors de l'insertion de trois lignes dont seule la colonne data1 est initialisée, la situation se présente comme ci-dessous :

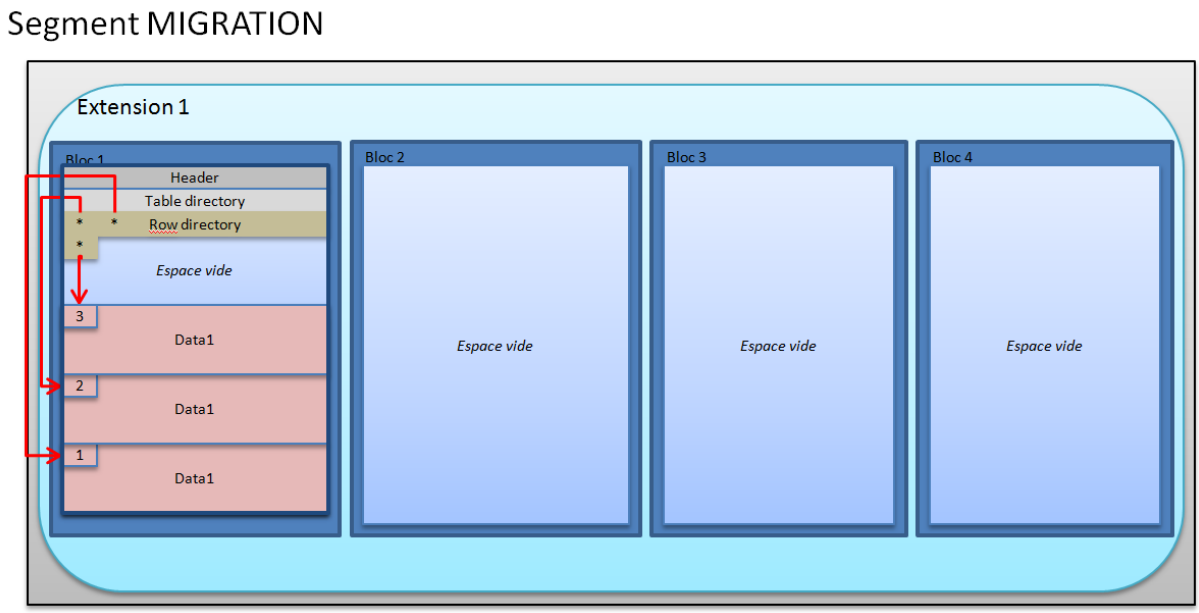


**Figure 4-1 : Illustration d'un segment après trois insertions**

Il reste un peu plus de 2000 octets dans le bloc 1, mais pas suffisamment d'espace pour faire la mise à jour d'une ligne si 4000 caractères de plus doivent être stockés, soit:

Update MIGRATION set data2 = '*d1*', data3 = '*d1*' where id = 1;

Lors de la mise à jour, il n'est pas possible de faire l'insertion dans le bloc courant. Par conséquent, le moteur va migrer la ligne vers le premier bloc disponible. Afin d'éviter la restructuration de ses références, Oracle laisse un pointeur indiquant l'emplacement du nouvel enregistrement.

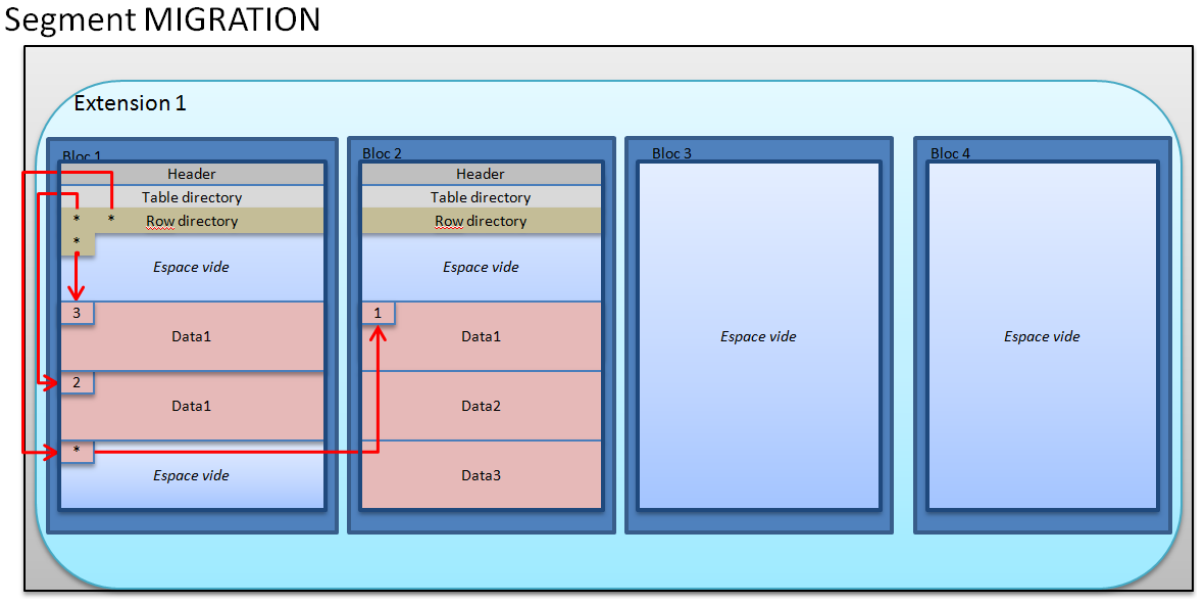


**Figure 4-2 : Migration du premier enregistrement**

### 4.3.3 Chaînage

Le chaînage est l'action réalisée par le *SGDB* lorsqu'une ligne possède une taille plus élevée que la taille d'un bloc [10]. Oracle va placer ce nouvel enregistrement dans plusieurs blocs distincts. L'emplacement de la suite de l'enregistrement contenu dans le bloc est indiqué par un pointeur. Celui-ci désigne où se poursuit l'enregistrement.

En reprenant la même situation que celle utilisée dans le point précédent et en ajoutant un nouvel enregistrement d'une taille de 10Ko, Oracle place le nouvel enregistrement dans le bloc 3 et la partie qui ne peut pas être contenue par le bloc 3 sera placée dans le bloc 4. La situation est décrite dans le graphique ci-dessous.

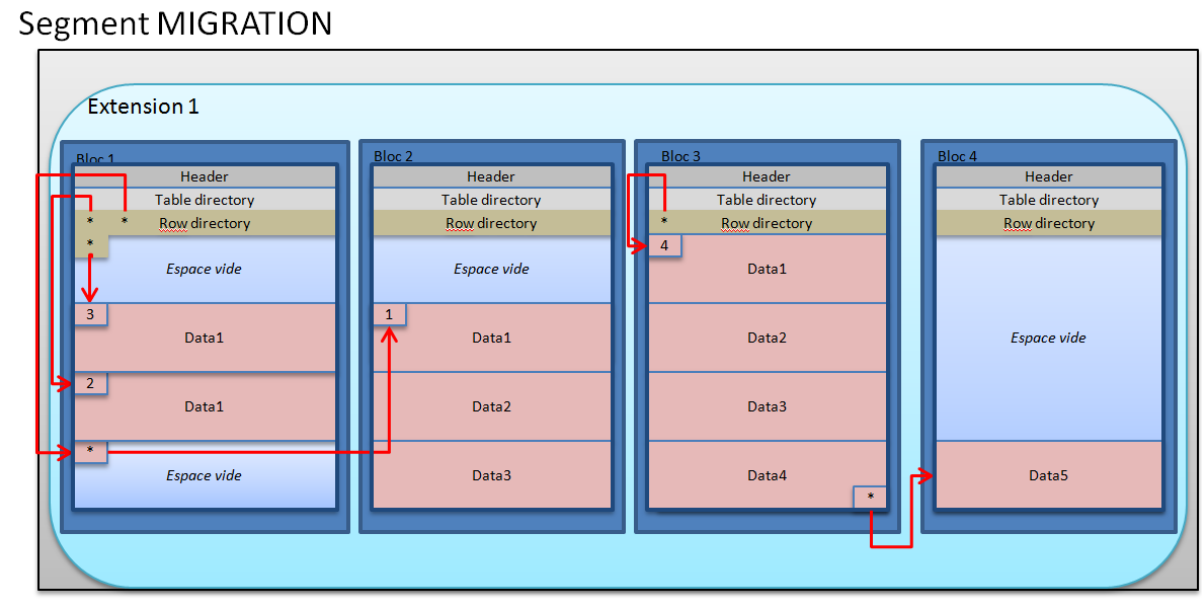


**Figure 4-3 : Chaînage du quatrième enregistrement**

Oracle stocke les grands objets au sein des structures différentes qui leur sont dédiées. Dès lors, chaque ligne qui possède un objet de grande taille subit ce phénomène de chaînage lorsqu'on y accède.

### 4.3.4 Organisation aléatoire

**Ordre logique des enregistrements au sein de la table**

Dans le cadre d'une table *HOT*, les données sont placées en vrac, c'est-à-dire, là où le *SGBD* trouve de la place sur ce segment. Si les enregistrements sont insérés de manière aléatoire, il devient très coûteux de les lire de manière séquentielle. Cette notion d'ordre logique des données au sein d'un segment, par rapport à l'ordre de l'index, est calculée à l'aide du *CF* (cf. section 3.3.1.2 ) pour une table *HOT*.

**Rupture de séquence des blocs index**

Au niveau d'un index, les insertions aléatoires provoquent un phénomène similaire : les données sont classées de manière séquentielle au niveau logique, mais non séquentielle au niveau physique sur l'espace de stockage. Cette notion est définie au sein de l'annexe H de l'ouvrage « Bases de données » [3]. Au sein du moteur Oracle, chaque division de type 50/50 ou IOT provoque une rupture de la séquence physique des blocs. En effet, le nouveau bloc n'est pas contigu avec celui qui vient d'être divisé : il est hors de la séquence physique. Lorsque le segment est constitué d'un grand nombre d'extensions, le temps d'accès au nouveau bloc équivaut au temps d'une lecture aléatoire. Pour une rupture, il faut théoriquement réaliser deux lectures aléatoires : une pour accéder au fragment distant et une pour revenir à la suite de la lecture séquentielle (opération coûteuse). Le coût en termes de temps pour ce phénomène est défini dans la section 5.6

## 4.4 Synthèse

La problématique de la dégradation des performances liée à la fragmentation est maintenant définie. La fragmentation est un phénomène physique et logique lié, d'une part, au système d'exploitation et, d'autre part, à la manipulation des données. Elle est favorisée au sein d'Oracle par sa structure logique mais également par les opérations de migrations et de chaînages et enfin, par les insertions désordonnées.

Ce phénomène a tendance à diminuer les performances du *SGBD*. L'objectif de ce dernier étant de fournir une structure efficiente pour le stockage et l'accès aux données, il est important d'en comprendre les causes afin de réduire l'apparition de la fragmentation.

# 5 Calcul des temps de lecture

## 5.1 Préambule

La section précédente expose une problématique liée à Oracle : la fragmentation.

Il s'agit maintenant de définir plusieurs notions afin de mettre en place des modèles permettant de calculer le coût d'accès à l'information.

Sur quels principes se basent ces modèles ? Quels sont les différents scénarios pertinents à envisager pour calculer les temps d'accès aux données ? Comment modéliser le phénomène de fragmentation ?

L'ensemble de ce chapitre s'appuie sur les développements établis par Jean-Luc Hainaut dans son livre « *Bases de données. Concepts, utilisation et développement* » [3]. Une partie de la méthodologie énoncée dans cet ouvrage est également reprise.

## 5.2 Définition de la méthode d'analyse

Les sections suivantes modélisent deux types d'objets et proposent une étude de cas empirique. L'analyse portera, d'abord, sur des tables *HOT* comportant un index *B-tree* sur leur clé primaire et, ensuite, sur des tables *IOT*. Ces deux analyses seront axées sur différents cas d'utilisation classiques des tables :

- Les tables en lecture seule (*read only*) ;
- Les tables sur lesquelles seuls des ajouts de lignes (*append only*) sont réalisés. Ce cas sera décomposé en deux cas de figure : l'ajout de lignes ordonnées et l'ajout de lignes désordonnées ;
- Les tables subissant des opérations d'insertion et de suppression, de manière ordonnée et désordonnée.

Dans chacune de ces situations, cette analyse étudie l'efficience des requêtes de différents types : celles dotées d'une grande sélectivité, celles avec une faible sélectivité et celles sur un seul enregistrement. De plus, une modélisation mathématique est réalisée pour chacune de ces situations.

Les deux études sont réalisées en se basant sur la même situation initiale.

## 5.3 Postulats initiaux

Cette section reprend un ensemble de postulats sur un disque définit par monsieur Hainaut dans son livre « Base de données » ainsi que sa méthodologie [3] afin de pouvoir étudier les impacts de la fragmentation sur les structures disponibles au sein d'Oracle. Les différentes valeurs obtenues permettent de réaliser des calculs de temps d'accès pour un disque avec ces spécifications. Cette étude utilise les valeurs d'un disque de 7200 rpm, soit un temps de rotation du disque de 8,33 ms ($\mathbf{t_{rot}}$). Le temps d'accès à une piste est la somme du temps de déplacement du bras de lecture vers le cylindre : $\mathbf{t_{db}}$ (entre 2,5 et 15 ms retenons une moyenne de 8 ms), le temps de commutation de la tête de lecture $\mathbf{t_{ctl}}$ (N.B. Il s'agit d'une opération électronique d'un temps négligeable), du temps d'attente de la piste, soit en moyenne, une demi-rotation : $\mathbf{t_{dr}}$ (4,17 ms pour un disque de 7200 rpm) et du temps de transfert du contenu du secteur vers la mémoire centrale : $\mathbf{t_{tr}}$ (0,074 ms) [3].

Le temps d'accès à une adresse en lecture aléatoire ($\mathbf{t_{la1}}$) est la somme des quatre temps définis ci-dessus :

$$\mathbf{t_{la1} = t_{db} + t_{ctl} + t_{dr} + t_{tr} = 8 + 0 + 4{,}17 + 0{,}074\ ms = 12{,}3\ ms}$$

Il est possible d'envisager un scénario intermédiaire en intégrant la mémoire tampon dans le temps de calcul (opération de lecture anticipée). Dès lors, l'opération revient à lire un enregistrement puis à réaliser le transfert des blocs suivants sur le disque. L'auteur signale dans son livre [3] que le coût total d'une lecture anticipée dépend du nombre de pages chargé par cette technique. En prenant un exemple de lecture anticipée de 112 blocs, ce qui représente environ la capacité d'une piste du disque dur (N.B. Cet exemple suit la description de la lecture anticipée tel que défini par monsieur Hainaut dans son livre), l'opération correspond à la somme du temps de lecture de la première page (soit 12,3 ms) et du temps de transfert des 111 pages suivantes (soit 111/112 x 8,33 = 8,26 ms). Par conséquent, le coût total est 20,56 ms pour 112 pages, soit un temps de lecture de bloc anticipé de **:**

$\mathbf{t_{lba}}$ **= 0,184** ms par page.

## 5.4 Lecture séquentielle

Soit un fichier contenant un *tablespace*, qui lui-même contient un segment constitué de $\mathbf{N_{Ec}}$ groupes d'extensions contigües. Ces dernières comportent un nombre croissant de blocs par extension.

$\mathbf{t_{lsbc}}$ - Lecture d'une courte séquence $\mathbf{N_b}$ de blocs contenue dans une extension. Par définition, ces $\mathbf{N_b}$ blocs sont consécutifs sur le disque étant donné qu'ils appartiennent au même segment. Le temps d'accès au premier bloc est $\mathbf{t_{la1}}$ et la lecture des pages suivantes est réduite à leur temps de transfert, soit $\mathbf{t_{tr}}$. Le temps de lecture d'un segment est donc:

$$\mathbf{t_{lsbc} = t_{la1} + (N_b - 1) \times t_{tr}}$$

$\mathbf{t_{lsbl}}$ - Lecture d'une séquence $\mathbf{N_b}$ de blocs contenue sur $\mathbf{N_{Ec}}$ groupes d'extensions contigües. Par définition, les blocs de chaque extension sont consécutifs sur le disque. Par contre, les extensions ne sont pas mutuellement consécutives. Le temps d'accès du premier bloc du premier segment est de $\mathbf{t_{la1}}$. Par la suite, à chaque lecture additionnelle d'un segment, il faut repositionner les têtes de lectures sur le premier bloc du segment. Cette opération de repositionnement est effectuée $\mathbf{N_{Ec}}$ fois. Le temps de lecture séquentielle d'une large séquence de blocs (placée sur plusieurs segments) est donc:

$$\mathbf{t_{lsbl} = N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}}$$

Remarque: $\mathbf{N_{Ec}}$ est le nombre de groupe d'extensions contigües sur le disque. Une extension seule incrémente également ce nombre.

**Lecture séquentielle d'un segment**

Cette analyse calcule le temps nécessaire pour lire un segment. Elle ne tient pas compte du temps d'accès au catalogue (N.B. il est déjà chargé en mémoire la majorité du temps). Le coût de la lecture de ce segment est donc celui de la lecture des blocs sur le disque. Cette analyse dépendra de 2 éléments:

- Le segment est-il d'un seul tenant ou est-t-il fragmenté?
- Le disque est-il réservé pour le processus de lecture ou est-il partagé entre plusieurs processus concurrents?

Dans un premier temps, cette analyse suppose que le segment n'est pas fragmenté et donc que l'ensemble des extensions soient contigües sur le disque. Il s'agit du meilleur scénario, les autres scénarios sont analysés par la suite :

Scénario 1 : Le disque est dédié. Il est donc possible de charger les blocs d'une seule lecture séquentielle. Le coût est, par conséquent, équivalent à la lecture d'une séquence de blocs contigus.

$$t_{lss1} = t_{la1} + (N_b - 1) \times t_{tr}$$

En partant du postulat que la taille du fichier est grande, le coût de l'accès initial peut être considéré comme insignifiant par rapport à la lecture de l'ensemble du fichier. Dès lors, il est possible de simplifier la formule de la manière suivante:

$$t_{lss1} = N_b \times t_{tr}$$

Scénario 2 : Le disque est partagé et permet la lecture anticipée. Cette situation est une situation courante, il s'agit d'une suite de lectures anticipées:

$$t_{lss2} = N_b \times t_{lba}$$

Scénario 3 : Le disque est fortement partagé sans possibilité de lecture anticipée. Il s'agit de la configuration la moins favorable, elle engendre une série de lectures aléatoires.

$$t_{lss3} = N_b \times t_{la1}$$

Au sein d'Oracle, deux phénomènes de fragmentation physique vont coexister :

- D'une part, la fragmentation du fichier. Chaque *SE* gère les fichiers d'une manière qui lui est propre et peut entraîner que le fichier ne soit pas d'un seul tenant sur son disque ;
- D'autre part, le segment au sein du *tablespace* peut être constitué d'extensions non contigües.

## 5.5 Influence de la fragmentation

Le nombre de fragment (**q** fragments) est le nombre de groupes d'extensions qui sont contigües ($N_{Ec}$). Typiquement, lors d'un chargement initial, toutes les extensions sont contigües, mais ce n'est pas le cas lors d'un usage partagé d'une instance Oracle, soit :

$$q = N_{Ec}$$

Les scénarios décrits dans le point précédent ne sont pas affectés de la même manière par un fichier fragmenté. Le scénario 1 est le plus impacté par le phénomène de fragmentation. En effet, les autres scénarios doivent, par définition, replacer les têtes de lecture après un accès partagé.

La formule du scénario 1 en cas de fragmentation devient alors:

$$t_{lss4} = q \times (t_{la1} - t_{tr}) + N_b \times t_{tr}$$

À partir de ces formules, il est possible de calculer la dégradation liée au degré de fragmentation d'un fichier. Il s'agit du rapport entre $t_{lss4}$ et $t_{lss1}$, soit:

$$D = t_{lss4} / t_{lss1} = ( q \times (t_{la1} - t_{tr}) + (N_b \times t_{tr} ) ) / ( N_b \times t_{tr} )$$

$$D = q \times (t_{la1} - t_{tr}) + 1$$

## 5.6 Rupture de séquence

La section 4.3.4 décrit deux phénomènes liés à l'ordonnancement des données. Un de ces deux phénomènes est la rupture de séquence. Il est possible de connaître la perte de temps lié à la rupture de séquence. Soit **hs** le nombre de blocs qui sont hors de la séquence physique.

Sans lecture anticipée, il faut réaliser deux $t_{la1}$ par rupture de séquence : $t_{la1}$ pour accéder bloc qui est hors de la séquence et $t_{la1}$ pour revenir dans la séquence pour continuer la lecture.

Avec lecture anticipée, il faut réaliser une $t_{la1}$ car le reste de la séquence est chargé dans un tampon avant de lire le bloc hors séquence.

Le taux de rupture ($\tau_r$) est la proportion de blocs qui sont hors séquence et est défini tel que :

$$\tau_r = hs / N_b$$

$\tau_r = 0$ lorsque la séquence de blocs forme un tout ordonné, sans rupture et $\tau_r = 1$ lorsque tous les blocs sont disposés aléatoirement sur le support physique.

Soit $t_{lhs}$ étant le temps de lecture d'une page hors séquence, le temps de lecture d'une séquence d'enregistrements est :

$$t_{lsbl} = ( 1 - \tau_r ) N_b \times t_{lba} + \tau_r \times t_{lhs}$$

La valeur de $t_{lhs}$ varie en fonction de plusieurs facteurs tels que le nombre d'extensions ou encore le taux de partage du disque. Dans la configuration la moins favorable $t_{lhs} = t_{la1}$.

## 5.7 Synthèse

Les différentes études de cas à réaliser sont maintenant définies. Le comportement des performances d'accès aux structures de type *HOT* et *IOT* est étudié dans 3 cas:

- L'accès en lecture de données figées (table en lecture seule) ;
- L'accès à un nombre d'enregistrements grandissant (*table append* avec insertions ordonnées et désordonnées) ;
- L'accès à des enregistrements au sein d'une table qui subit des insertions et des suppressions ordonnées et désordonnées.

Plusieurs postulats relatifs à l'utilisation du disque sont également définis.

De plus, les temps de lecture d'un segment sont modélisés dans plusieurs scénarios, il s'agit des temps de lecture :

- D'un segment sur un disque dédié, composé d'extensions contiguës;
- D'un segment sur un disque dédié, composé d'extensions non contiguës ;
- D'un segment placé sur un disque partagé qui permet les lectures anticipées;
- D'un segment placé sur un disque fortement partagé ne permettant pas de lectures anticipées.

Enfin, cette section permet, grâce à une modélisation, de comprendre la dégradation des performances liées à la fragmentation d'un fichier.

# 6 Analyse d'une table *HOT*

## 6.1 Préambule

La section précédente présente différents modèles qui permettent de calculer les temps d'accès à un segment en fonction de différents critères d'accès ou de partage des ressources.

Maintenant, la structure de type *HOT* accompagnée de son index, peut être étudiée au travers de différents cas pratiques.

Mais comment analyser ces structures? Quelles sont les mesures possibles pour calculer un objet de type *HOT* et pour un index qui lui est lié? Quel est le temps d'accès à un enregistrement? Il est également intéressant d'analyser et de se questionner sur les temps nécessaires à la réalisation des différentes opérations *DML* et sur l'impact que peut avoir le taux de remplissage de la table ou de l'index. L'organisation dans une table *HOT* est-elle indépendante des performances d'accès à la table ? Enfin, comment se comporte celle-ci face aux situations définies dans la section 5 ?

## 6.2 Volume d'une table *HOT*

Le volume d'une table *HOT* est la somme des volumes objets qui la composent. Dans les exemples proposés, la table *HOT* dispose d'un index qui lui est lié. Par conséquent, la taille totale d'une table *HOT* ($\mathbf{N_{b_{HOT}}}$) correspond à la somme du nombre de blocs qui constituent la table ($\mathbf{N_{bt}}$) et du nombre de blocs qui constituent l'index ($\mathbf{N_{bi}}$) soit:

$$\mathbf{N_{b_{HOT}} = N_{bt} + N_{bi}}$$

**Calcul de la taille d'un table *HOT* ($\mathbf{N_{bt}}$)**

La table contient $\mathbf{N_e}$ enregistrements de taille $\mathbf{L_e}$ stockés dans les $\mathbf{N_{bt}}$ blocs alloués à la table. Ces blocs disposent d'une taille disponible pour les enregistrements qui est fixe ($\mathbf{L_b}$). $\boldsymbol{\tau_b}$ représente le taux de remplissage des blocs de la table à un moment donné. À partir de ces paramètres, il est possible d'obtenir $\mathbf{M_{epbHOT}}$ et $\mathbf{N_{epbHOT}}$ qui représentent respectivement le nombre maximum et le nombre moyen d'enregistrements par bloc de la table.

$$\mathbf{M_{epbHOT} = \lfloor L_b / L_e \rfloor}$$

$$\mathbf{N_{epbHOT} = \tau_b \times M_{epbHOT}}$$

Partant de ces notions, le nombre de blocs qui composent une table *HOT* peut être défini avec la formule suivante :

$$\mathbf{N_{bt} = \lceil N_e / N_{epbHOT} \rceil}$$

**Calcul de la taille d'un index ($\mathbf{N_{bi}}$)**

$\boldsymbol{\tau_i}$ représente le taux d'occupation moyen des pages d'index à un moment donné. La longueur d'une entrée dans l'index est $\mathbf{L_i}$. Elle correspond à la somme de la longueur du champ clé $\mathbf{L_c}$ et de la longueur du *RowID* correspondant $\mathbf{L_{rid}}$. Par conséquent, le nombre d'entrées maximum se calcule à l'aide de la formule suivante:

$$\mathbf{L_i = L_c + L_{rid}}$$

Depuis Oracle 8, La taille $L_{rid}$ est de 10 octets.

Il est possible d'obtenir $M_{ipb}$ et $N_{ipb}$, respectivement le nombre maximum et le nombre moyen d'entrées par bloc de l'index.

$$M_{ipb} = \lfloor L_b / L_i \rfloor$$

$$N_{ipb} = \tau_i \times M_{ipb}$$

En règle générale, la taille de l'index est nettement plus faible que la taille de la table. De plus, il apparaît que la taille du dernier niveau est largement dominante soit $N_{bi}(n)$. Ce niveau contient une entrée par enregistrement contenu dans la table. La formule peut donc être simplifiée :

$$N_{bi} \approx N_{bi}(n) = \lceil N_e / N_{ipb} \rceil$$

**Calcul du nombre de niveaux de l'index n**

Chaque niveau d'index comporte $N_{ipb}$ plus de blocs que le précédent.

$$N_{bi}(n) = \lceil N_e / N_{ipb} \rceil \approx N_{ipb}^{n-1}$$

qui peut être simplifié en:

$$N_e \approx N_{ipb}^{n}$$

La hauteur de l'index (**H**) peut être définie tel quel :

$$H = n = \lceil \log_{Nipb} N_e \rceil = \lceil \log_b N_e / \log_b N_{ipb} \rceil$$

## 6.3 Temps de lecture

Lors de la lecture d'une ligne en passant par un index, Oracle parcourt celui-ci afin de trouver le bloc « feuille » correspondant aux critères, puis lit le bloc correspondant dans la table.

### 6.3.1 Temps de lecture d'une séquence d'enregistrements

Pour une table jeune qui n'a pas encore subi de modification (migration et chaînage), le temps de lecture de la table correspond au temps nécessaire à la lecture séquentielle des $N_b$ blocs contenus au sein des $N_{Ec}$ extensions qui la composent, soit :

$$t_{lsbl} = N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$$

Par la suite, chaque phénomène de chaînage ($N_{ch}$) ou de migration ($N_{mig}$) provoque une lecture aléatoire supplémentaire lors de l'accès à un enregistrement. Dès lors, le temps de lecture de la table devient :

$$t_{lsbl} = (N_E + N_{ch} + N_{mig}) \times t_{la1} + (N_b - N_E) \times t_{tr}$$

### 6.3.2 Temps de lecture d'un enregistrement par l'index

Au niveau physique, il s'agit de plusieurs lectures aléatoires engendrées par la lecture du bloc « racine », puis du/des bloc(s) « branche », puis du bloc « feuille » et enfin, du bloc qui contient l'enregistrement dans la table. Le temps de lecture d'un enregistrement est la somme du temps de

lecture de l'index ($t_{lci}$) et du temps de lecture du bloc correspondant dans la table ($t_{lct}$) (soit 1 si les phénomènes de chaînage et de migration ne sont pas pris en considération).

$$t_{lci} = H \times t_{la1}$$

$$t_{lct} = t_{la1}$$

Remarque : Oracle définit **BLevel** comme étant la hauteur des blocs « branche ». La hauteur de l'arbre (**H**) est définie comme :

$$H = Blevel + 1$$

Le temps de lecture d'un enregistrement est donc de :

$$t_{lc} = t_{lci} + t_{lct}$$

ou $$t_{lc} = ( H + 1 ) \times t_{la1}$$

### 6.3.3 Temps de lecture d'une séquence d'enregistrements par l'index

Le temps de lecture séquentielle d'un segment a déjà été calculé, il s'agit de :

$$t_{lsbl} = N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$$

Cette mesure fournit le temps nécessaire à la lecture d'un segment, que ce soit un index ou une table.

Il est important de se souvenir qu'une table et son index sont deux objets distincts. Par conséquent, le parcours d'une table triée sur l'index comprend le temps de lecture de l'index et le temps de lecture de la table.

Dans le cas idéal, après un chargement ou lorsqu'une table est utilisée uniquement avec des insertions sur des valeurs de clés croissantes, le temps de lecture d'un intervalle de clés via l'index ($t_{lici}$) en fonction de la *sélectivité des requêtes* (***sel***) est la somme du temps de lecture de la première clé de l'index, du temps de transfert des blocs de l'index qui suivent séquentiellement, du temps de lecture du bloc de la table lié à la première clé d'index et du temps de transfert des blocs de la table qui suivent séquentiellement :

$$t_{lici} = t_{lci} + (N_{bi}-1) \times sel \times t_{tr} + t_{la1} + (N_{bt} -1) \times sel \times t_{tr}$$

Si l'organisation des données au sein de la table n'est pas la même que celle d'un index, le calcul du temps de lecture est lié au *clustering factor*. Le moteur Oracle utilise principalement le *CF* combiné à la *sélectivité des requêtes* afin de définir une estimation du coût de lecture d'une suite d'enregistrements dont l'accès est réalisé à l'aide de l'index [8].

$$CF \times sel = \text{coût accès estimé via index}$$

Sans aucun mécanisme de mise en cache, le calcul du temps de lecture devient:

$$t_{lici} = t_{lci} + (N_{bi} -1) \times sel \times t_{tr} + CF \times sel \times t_{la1}$$

La formule ci-dessus, avec une valeur de ***CF*** élevée, illustre la situation la moins favorable. En intégrant la mémoire tampon dans le temps de calcul, l'opération revient à une lecture anticipée.

Avec un mécanisme de tampon, le temps de lecture devient :

$$t_{lici} = t_{lci} + (N_{bi} - 1) \times sel \times t_{tr} + CF \times sel \times t_{lba}$$

Lors de l'exécution de chaque requête, le moteur Oracle estime les coûts à l'aide du *Query Optimizer,* des différentes manières de réaliser l'opération. Si la lecture des enregistrements, en passant par l'index, lui semble plus coûteuse que la lecture séquentielle de l'ensemble du segment de la table suivi de son tri en mémoire, alors Oracle n'utilise pas l'index pour accéder aux enregistrements.

La formule suivante fournit le taux d'organisation de la table par rapport à l'ordre de l'index :

$$\text{Taux d'organisation des enregistrements} = 1 - | CF - N_b | / ( N_e - N_b )$$

En effet, si le *CF* est égal au nombre de blocs constituant la table, cela signifie que les enregistrements sont parfaitement organisés au sein de la table. Par conséquent, il est possible de réaliser une lecture de $N_{epbHOT}$ enregistrements par lecture de bloc de la table. Le taux d'organisation des enregistrements est de 1.

Si le *CF* est égal au nombre d'enregistrements, cela signifie que les enregistrements sont totalement désorganisés. Par conséquent, chaque enregistrement demande la lecture d'un nouveau bloc au sein de la table. Le taux d'organisation des enregistrements est de 0.

### 6.3.4 Calcul du temps des opérations sur une table *HOT*

Pour chacune des opérations décrites dans les points ci-dessous, le temps d'accès à l'enregistrement et le temps de modification est pris en compte. Une opération comporte deux étapes : La première consiste à localiser le positionnement de l'enregistrement, la seconde est l'opération proprement dite.

**Temps d'ajout d'un enregistrement**

Lors de l'insertion, il faut accéder au bloc correspondant dans l'index ($t_{lci}$) puis réécrire ce bloc. Si le bloc ne peut fournir l'espace nécessaire, il faut réaliser une opération de division. Il existe donc trois scénarios:

- Sans division :

  Il faut réécrire le bloc « feuille » une fois mis à jour ($t_{la1}$) puis enregistrer le bloc au sein de la table ($t_{la1}$) soit :

  $$t_{ins} = t_{lci} + 2 \times t_{la1}$$

- Soit 50/50 :

  Dans ce cas, il faut obtenir l'adresse d'un bloc vide. Oracle le fournit via un mécanisme souvent chargé en mémoire cache, il s'agit du *freelist*. Ensuite, les deux blocs contenant les enregistrements sont réécrits ($2 \times t_{la1}$). Il faut également mettre à jour le bloc « branche » correspondant ($t_{la1}$). De plus, la création d'un nouveau bloc nécessite la mise à jour du pointeur précédent du bloc suivant la division ($t_{la1}$). Enfin, il faut enregistrer la ligne au sein de la table ($t_{la1}$).

  $$t_{ins50} = t_{lci} + 5 \times t_{la1}$$

- Soit 90/10 :

  Auquel cas, il faut obtenir l'adresse d'un nouveau bloc et écrire le nouvel enregistrement au sein de celui-ci ($\mathbf{t_{la1}}$). Ensuite, il faut mettre à jour le pointeur contenu dans le bloc précédent ($\mathbf{t_{la1}}$). Puis, il faut mettre à jour le bloc « branche » correspondant ($\mathbf{t_{la1}}$). Enfin, enregistrer la ligne au sein de la table ($\mathbf{t_{la1}}$).

  $$\mathbf{t_{ins90} = t_{lci} + 4 \times t_{la1}}$$

Les deux types de division ont une certaine probabilité de se produire. Elles sont représentées de la manière suivante :

$\boldsymbol{\tau_{50}}$ est la probabilité d'une division 50/50 et $\boldsymbol{\tau_{90}}$ est la probabilité d'une division 90/10.

Lors d'insertions de clés ordonnées et croissantes, la technique utilisée sera celle du 90/10 où $\boldsymbol{\tau_{90} = 1 / M_{ipb}}$

Lors d'insertions aléatoires de clés, la technique de division utilisée sera principalement celle du 50/50 où

$\boldsymbol{\tau_{50} = 2 / M_{ipb}}$ et $\boldsymbol{\tau_{90}}$ est insignifiant.

Les divisions ont tendance à augmenter significativement le taux de rupture d'une séquence de données. Chaque rupture requiert un accès supplémentaire lors de la lecture séquentielle de l'ensemble des données.

**Temps de suppression d'un enregistrement**

Lors de la suppression d'un enregistrement, Oracle marque la ligne comme « supprimée » à l'aide d'un drapeau et effectue la même opération au niveau de l'index. Le coût de l'opération est celui de l'accès à deux pages, l'index et la table, soit $\mathbf{2 \times t_{la1}}$. Les pages concernées ne seront réorganisées qu'en cas de nécessité lors d'un nouvel ajout au sein de ce bloc.

$$\mathbf{t_{sup} = t_{lci} + 2 \times t_{la1}}$$

**Temps de modification d'un enregistrement**

La modification d'un enregistrement ($\mathbf{t_{mod}}$) dure $\mathbf{t_{la1}}$. Le moteur du *SGBD* réécrit le bloc modifié. En cas de migration, il écrit sur deux blocs, soit $\mathbf{2 \times t_{la1}}$. Lors de la modification d'une clé liée à un enregistrement, le *SGBD* réalise une suppression de la clé au niveau de l'index, puis effectue un ajout de la nouvelle clé au sein de cet index. Dans le meilleur des cas, si la nouvelle clé peut être placée au sein du même bloc, l'opération dure $\mathbf{t_{la1}}$ (N.B. il s'agit par exemple d'un changement d'orthographe d'un mot: DUPONT deviendrait DUPOND). Dans le cas le moins favorable, il faut noter l'enregistrement comme supprimé, l'ajouter dans un autre bloc de l'index, et enfin, modifier le bloc correspondant au sein de la table, soit $\mathbf{3 \times t_{la1}}$. Par conséquent, l'opération de modification dure entre $\mathbf{t_{la1}}$ et $\mathbf{3 \times t_{la1}}$.

$$\mathbf{t_{lci} + t_{la1} < t_{mod} < t_{lci} + 2 \times t_{la1}}$$

Temps de modification de la clé primaire :

$$\mathbf{t_{lci} + 2 \times t_{la1} < t_{mod} < t_{lci} + 3 \times t_{la1}}$$

## 6.4 Synthèse des calculs

$N_{bHOT}=N_{bt}+N_{bi}$

$M_{epbHOT} = \lfloor L_b / L_e \rfloor$ $\quad$ $M_{ipb} = \lfloor L_b / L_i \rfloor$

$N_{epbHOT} = \lfloor \tau_b \times M_{epbHOT} \rfloor$ $\quad$ $N_{ipb} = \lfloor \tau_i \times M_{ipb} \rfloor$

$N_{bt} = \lceil N_e / N_{epbHOT} \rceil$ $\quad$ $N_{bi} \approx N_{bi}(n) = \lceil N_{epbHOT} / N_{ipb} \rceil$

$L_i = L_c + L_{rid}$

Temps de lecture d'une séquence d'enregistrements

$t_{lsbl} = (N_{Ec} + N_{ch} + N_{mig}) \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$

Temps de lecture d'un enregistrement

$t_{lc} = t_{lci} + t_{lct}$

## 6.5 Impact du taux de remplissage de l'index et du *Clustering Factor* sur les L/E lors de la lecture d'une table

Les lignes qui suivent montrent l'impact de différentes notions applicables sur la table *HOT* et son index. Ces notions sont le taux de remplissage de l'index ainsi que le taux d'organisation des lignes dans la table *HOT* en fonction de l'index (N.B. Cette notion est quantifiée en Oracle à l'aide du *Clustering Factor* (*CF*)). Cet exemple illustre les notions de table *HOT*, index et CF définies dans la section 3

Le nombre de lectures et d'écritures (L/E) nécessaires afin de réaliser une requête, varie en fonction du taux de remplissage de l'index et du *CF*. Il est possible d'extrapoler le nombre de L/E en fonction de ces deux valeurs.

En ce qui concerne le taux de remplissage de l'index, dans le meilleur cas possible, il est de 100%. À l'inverse, il est aussi envisageable de calculer le taux de remplissage théorique le plus défavorable. Il s'agit du cas où il est nécessaire d'ajouter une ligne dans chaque bloc déjà plein. Dans ce second cas, Oracle doit réaliser une division des blocs par la méthode 50/50. Par conséquent, l'index doit comporter deux fois plus de blocs pour contenir le même nombre de valeurs.

Le deuxième élément dont il convient de tenir compte est le *CF*.

Dans le meilleur des cas (*CF* faible), la sélectivité d'une requête multipliée par le nombre de blocs de la table permet d'obtenir une estimation du nombre de blocs nécessaires pour obtenir les lignes désirées.

Dans le cas le plus défavorable (*CF* élevé), les données sont totalement désorganisées au sein de la table, et le moteur Oracle doit lire l'ensemble des lignes en réalisant une nouvelle lecture pour chaque enregistrement.

Voici l'étude de cas particuliers se basant sur un index unique totalement rempli d'une part, et un index ayant subi le nombre maximum de divisions de blocs de type 50/50, d'autre part. De plus, chaque exemple sera analysé avec un *CF* idéal (table ordonnée) et un mauvais *CF* (table désordonnée).

Dans cet exemple, la taille des blocs est de 16Ko. Sur cet espace, seuls 16000 octets sont alloués pour le stockage des données. Le reste du bloc est réservé pour stocker les informations de l'entête (cf. section 2.2.2 ).

Le tableau est composé de :

- 1 000 000 de lignes ;
- 10 000 blocs de données (soit 100 lignes par bloc) ;
- Chaque enregistrement occupe un espace de 160 octets.

Chaque valeur de clé de l'index occupe un espace de 6 octets, soit une longueur totale par clé de 16 octets :

$$L_i = L_c + L_{rid}$$

$$L_i = 6 \text{ octets} + 10 \text{ octets}$$

$$L_i = 16 \text{ octets}$$

Dans la situation **(A)**, l'index est rempli à 100% et ses caractéristiques sont :

- 1051 blocs d'index ;
- Hauteur de l'index est de 3 ;
- Le nombre de blocs branche inclus dans l'index est 50 ;
- Le nombre de blocs « feuille » qui composent l'index est 1000 (soit 1000 clés d'enregistrements par bloc).

Dans la situation **(B)**, l'index est rempli à 50% et ses caractéristiques sont :

- 2101 blocs d'index ;
- Hauteur de l'index est de 3 ;
- Le nombre de blocs « branche » inclus dans l'index est le double, soit 100 ;
- Le nombre de blocs « feuille » qui composent l'index est également le double soit 2000 (soit 500 clés d'enregistrements par bloc).

(BR=Bloc « racine » ; BB=Bloc « branche » ; BF=Bloc « feuille » ; BT= Bloc Table)

Les cadres ci-dessous illustrent le calcul des coûts de différentes requêtes sur un modèle particulier. Chacune de celles-ci dispose d'une sélectivité différente afin d'observer son impact en termes de L/E. Dans chacun de ces cadres, la première estimation correspond au meilleur *CF*, et la seconde au *CF* le plus défavorable. De cette manière, les quatre possibilités envisagées sont expérimentées.

La modification du *CF* demande le rechargement de la table de manière ordonnée, il s'agit d'un processus très coûteux. L'exemple ci-dessous prend comme postulat de garder très peu de blocs en cache afin d'accentuer le coût de cette désorganisation. Dans tous les cas sélectionnant un champ au sein de la ligne, le *CF* a bien plus d'impact que le taux de remplissage de l'index. Pour compléter chaque exemple, le pourcentage de perte de performances dû au faible taux de remplissage de l'index est présenté.

*Requête sélectionnant une ligne :*
*CF idéal (chargement parfaitement ordonné)*
*Cas **(A) = 1 BR + 1 BB + 1 BF + 1 BT = 4 L/E***
*Cas **(B) = 1 BR + 1 BB + 1 BF + 1 BT = 4 L/E***

Si la hauteur de l'arbre est identique, le fait que l'index soit réparti sur un grand nombre de blocs ne change pas le nombre de lectures/écritures. Le *CF* n'a également aucune importance étant donné qu'il ne faut charger qu'un bloc de la table.

*Requête sélectionnant 1000 lignes de manière séquentielle (soit 0.1% de la table) :*
*CF idéal (chargement parfaitement ordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.001*1000 BF + 0.001*10000 BT = 13 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.001*2000 BF + 0.001*10000 BT = 14 L/E***
*Pire CF (chargement désordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.001*1000 BF + 1000 BT = 1003 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.001*2000 BF + 1000 BT = 1004 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de :

- *CF* idéal - 1 L/E soit 7,69% ;
- Pire *CF* - 1 L/E soit 0,1%.

*Requête sélectionnant 10000 lignes de manière séquentielle (soit 1% de la table) :*
*CF idéal (chargement parfaitement ordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.01*1000 BF + 0.01*10000 BT = 112 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.01*2000 BF + 0.01*10000 BT = 122 L/E***
*Pire CF (chargement désordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.01*1000 BF + 10000 BT = 10012 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.01*2000 BF + 10000 BT = 10022 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de :

- *CF* idéal - 10 L/E soit 8,19% ;
- Pire *CF* - 10 L/E soit 0,1%.

*Requête sélectionnant 100000 lignes de manière séquentielle (soit 10% de la table) :*
*CF idéal (chargement parfaitement ordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.1*1000 BF + 0.1*10000 BT = 1102 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.1*2000 BF + 0.1*10000 BT = 1202 L/E***
*Pire CF (chargement désordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 0.1*1000 BF + 100000 BT = 100102 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 0.1*2000 BF + 100000 BT = 100202 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de :

- *CF* idéal - 100 L/E soit 8,32% ;
- Pire *CF* - 100 L/E soit 0,1%.

*Requête sélectionnant 1000000 lignes de manière séquentielle* ***forçant*** *l'utilisation de l'index (soit 100% de la table) :*
*CF idéal (chargement parfaitement ordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 1*1000 BF + 1*10000 BT = 11002 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 1*2000 BF + 1*10000 BT = 12002 L/E***
*Pire CF (chargement désordonné)*
*Cas* ***(A) = 1 BR + 1 BB + 1*1000 BF + 1000000 BT = 1001002 L/E***
*Cas* ***(B) = 1 BR + 1 BB + 1*2000 BF + 1000000 BT = 1002002 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de :

- *CF* idéal - 1000 L/E soit 8,33%;
- Pire *CF* - 1000 L/E soit 0,1%.

**Synthèse des exemples**

Les exemples ci-dessus permettent d'aboutir à deux intuitions qui corroborent la description du CF (Cf. section 3.3.1.2 ) :

- Dans le cas d'une lecture séquentielle, le *CF* a plus d'impact sur les performances que l'état de l'index. En effet, généralement la taille de l'index est largement inférieure à celle de la table. Par conséquent, une dégradation de la taille de l'index a un impact limité. Par contre, si le CF est élevé, cela entraîne une hausse substantielle du nombre de lecture. Bien plus élevé que la perte de performance induite par un mauvais taux de remplissage de l'index.
- En fonction du *CF* : la perte de performance liée à un faible taux de remplissage est assez négligeable : 0,1% (Mauvais *CF*) ou est relativement conséquente : 8,33% (Bon *CF*).

Assez logiquement, la perte de performance d'un *full index scan* est d'environ 50% :

*Coût d'un full index scan:*
*Cas* ***(A) = 1 BR + 50 BB + 1*1000 BF = 1051 L/E***
*Cas* ***(B) = 1 BR + 100 BB + 1*2000 BF = 2101 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de 1050 L/E soit 49,98%.

## 6.6 Phénomène de perte de performances d'une table *HOT*

De manière générale, au fil de ce document, un certain nombre d'éléments participent à la perte de performances d'une table *HOT*. Il est possible d'associer ceux-ci soit à la table, soit à l'index correspondant. Les éléments influençant le temps d'accès à une table sont (Cf. sections 6.3 ) :

- sa taille : plus une table est grande, plus son coût de lecture sous les mêmes hypothèses est élevé ;
- sa fragmentation logique (le chaînage et la migration) : chaque occurrence de ces phénomènes entraîne une lecture additionnelle ;
- le taux de remplissage de ses blocs : plus ce taux est faible, plus le nombre de blocs à parcourir pour accéder à un ensemble d'enregistrements est élevé ;
- son chargement initial (ordonné ou en vrac) : moins les enregistrements sont ordonnés, plus les accès séquentiels entraînent des accès aléatoires ;
- son taux de modification (modification et suppression) : plus ce taux est élevé, plus les enregistrements se désordonnent au sein de la table ;

Les différents éléments ayant une incidence sur un index sont (Cf. sections 3.3.2 et 6.3.3 ) :

- sa taille : plus un index est grand, plus son coût de lecture sous les mêmes hypothèses est élevé ;
- son *CF :* plus celui-ci est élevé, plus le coût de lecture séquentiel de la table est élevé ;
- le taux de remplissage de ses blocs : plus ce taux est faible, plus le nombre de blocs à parcourir pour l'ensemble des clés d'index est élevé ;
- le taux de rupture des séquences blocs qui le compose : plus des divisions de type 50/50 sont réalisées, plus le taux de rupture des séquences est élevé.

Ces divers éléments permettent d'obtenir un indice de performance qui peut être attendu pour une table donnée.

### 6.6.1 Situation initiale

Cette section analyse une table de type *HOT* et son index associé. Lors de leur création, ces deux objets sont créés au sein d'un segment qui reçoit une extension afin de stocker les métadonnées leur correspondant.

Voici la structure de la table :

| COLUMN_NAME | DATA_TYPE | NULLABLE |
|---|---|---|
| IDENT_NUM | NUMBER(20,0) | No |
| DATEI_DAT | DATE | No |
| RANDOM_VARCHAR | VARCHAR2(1000 BYTE) | Yes |

Il s'agit d'une structure assez simple comportant un chiffre qui est l'identifiant, une date d'insertion ainsi qu'une chaîne de caractères de taille variable, qui est initialement chargée avec 500 caractères. Cette taille est choisie afin de favoriser les migrations de lignes lors des mises à jour. Le taux de remplissage initial des blocs, au niveau de la table, est de 80%. Il reste donc 20% réservés à la modification des entrées contenues dans un bloc de la table.

Dans cette configuration, la taille des blocs est de 8Ko. Dans ceux-ci, seuls 8000 octets sont réservés pour les données, le reste du bloc étant alloué à l'entête. La taille moyenne d'une entrée dans l'index est de 15 octets et la taille moyenne d'une ligne au sein de la table est de 511 octets.

Remarque : La taille d'un nombre stocké au sein d'Oracle est variable en fonction de sa valeur. La taille de la clé identifiant est de 2 octets lors des premières insertions et est de 7 octets lors des dernières insertions. Cette particularité fait que la taille de l'enregistrement et la taille de la clé d'index ne sont pas fixes. Lors des calculs de cette section, une valeur moyenne de cette taille est utilisée, ceci entraîne une légère différence entre les statistiques fournies par la simulation et le modèle défini en amont.

### 6.6.2 Table Append

Une table *Append* (Cf. section 5.2 ) est un objet qui subit uniquement des opérations d'insertion. Dans un premier temps, ce document analyse le comportement de ce type de table avec des ajouts ordonnés ou désordonnés. Ensuite, la partie suivante présente des tables qui subissent diverses opérations *Data Manipulation Language* (*DML*).

#### Ajout ordonné croissant

Le premier cas étudié est celui d'une table qui comporte une série d'insertions ordonnées et croissantes. Dans ce cas, la technique d'accroissement de l'index consiste en une série de divisions de type 90/10 (N.B. La section 3.3.2 rappelle que la technique nommée 90/10 est un fait une division 99/1). En effet, la Figure 6-1 montre que l'index obtient un taux d'utilisation proche des 100%. L'autre caractéristique intéressante concerne l'ordre des lignes dans la table qui correspond parfaitement à l'ordre d'insertion. Ceci entraîne que le *CF* est proche du nombre de blocs contenant les données de la table *HOT* (cf. *Le clustering factor*). Cette organisation permet d'avoir un accès optimisé aux enregistrements stockés au sein de la table tel que démontré dans la section 6.5 .

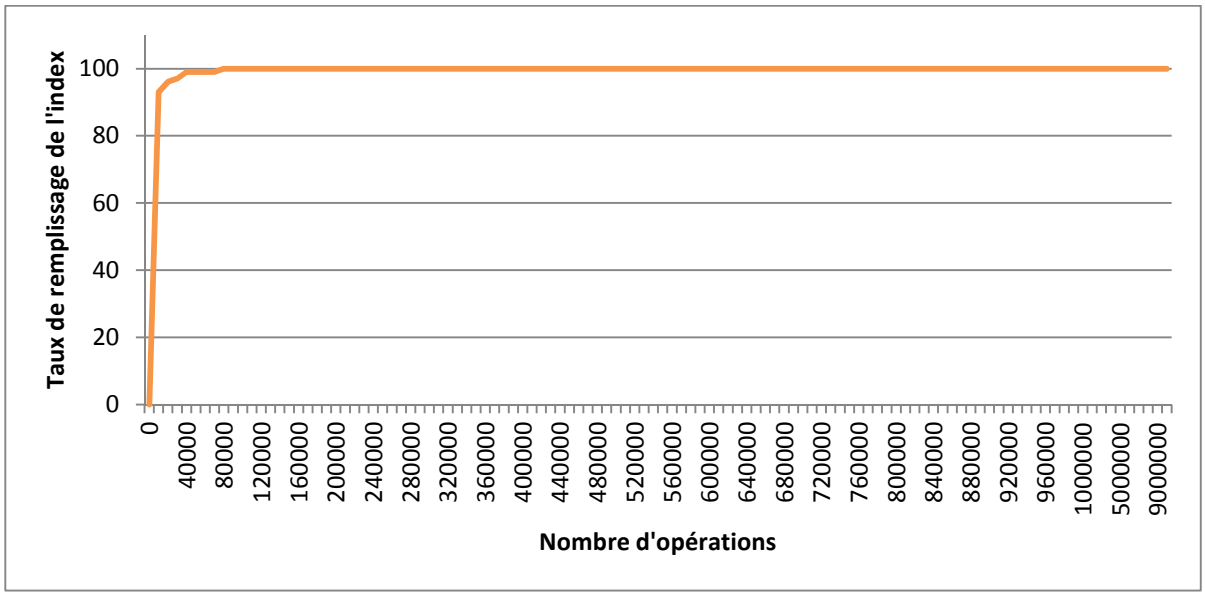


**Figure 6-1: Taux de remplissage de l'index (insertions ordonnées croissantes HOT)**

De plus, au niveau de la table, les insertions se réalisent également de manière séquentielle. Le graphique ci-dessous illustre le coût d'une lecture séquentielle de la table en utilisant l'index. Le rapport est de *CF* et le nombre de blocs qui constitue la table est de 1 ce qui constitue la situation idéale (Cf. *Le clustering factor* à la section 3.3.1.2 ), soit :

$$\frac{\mathbf{CF}}{\mathbf{Nbt}} = \frac{\mathbf{83361}}{\mathbf{83615}} \approx \mathbf{1}$$

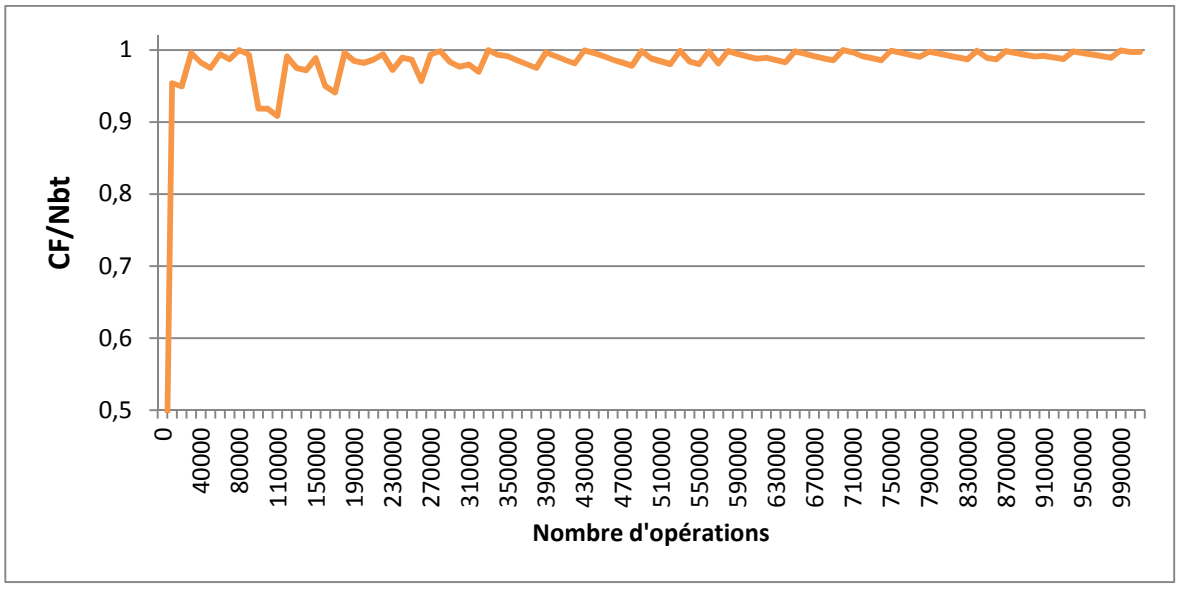


**Figure 6-2 : Observation du *Clustering Factor* (insertions ordonnées croissantes HOT)**

Les irrégularités du graphique ci-dessus trouvent leur origine dans deux phénomènes :

- Le *CF* peut être inférieur au nombre de blocs total de la table. Cela se produit lors de chaque extension du segment de la table *HOT* (Cf. section 3.3.1.2 ). Par conséquent, à chaque extension, le graphique affiche une irrégularité ;
- La taille des extensions varient. Au départ, chaque extension a une taille de 8 blocs. À mesure que la table s'agrandit, la taille des extensions allouées s'agrandit également. Cependant, la

taille des extensions est proportionnellement de plus en plus faible par rapport à la taille totale du segment (N.B. soit la somme des extensions qui le composent cf. la section 2.2.2 ). Ceci explique que les irrégularités vont en diminuant.

Au terme de cette simulation, les statistiques d'Oracle fournissent plusieurs valeurs :

**$N_{bt}$ = 83537 ; CF = 83352 ; $N_{bi}$ = 1920 ; $N_{bf}$ = 1875 ; $N_{bb}$ = 4 ; $L_i$ = 18,2899**

Remarque : Le nombre de blocs de la table et le nombre de blocs de l'index sont le nombre de blocs alloués via des extensions.

Il est possible de calculer cette valeur à partir du modèle établi préalablement.

**$N_e$ = 1000000**

**$M_{epbHOT} = \lfloor L_b / L_e \rfloor$ = 15 enregistrements par bloc**

**$N_{epbHOT} = \tau_b \times \lfloor L_b / L_e \rfloor = 0.8 \times \lfloor 8000 / 511 \rfloor$ = 12 enregistrements par bloc**

**$N_{bt} = \lceil N_e / N_{epbHOT} \rceil$ = 83334 blocs de la table *HOT***

La table occupe 83334 blocs. Ce qui correspond à la taille du CF, soit le nombre de blocs à lire pour parcourir la table de manière ordonnée sur l'index.

**$M_{ipb} = \lfloor L_b / L_i \rfloor$ = 437 clés par bloc**

**$N_{ipb} = \tau_i \times \lfloor L_b / L_i \rfloor$ = 222 clés par bloc**

**$N_{bi} \approx N_{bi}(n) = \lceil N_e / N_{ipb} \rceil$ = 1875 blocs de l'index**

En moyenne, le parcours des valeurs contenues au sein d'un bloc de l'index demandera l'accès à peu de blocs de la table car les enregistrements y sont placés de manière séquentielle.

Dans cet exemple, chaque bloc de l'index référence les clés comprises dans 44 blocs de la table :

**CF / $N_{bi}$ = 83334 / 1875 = 44 blocs de la table**

Au vu du mode d'insertion, dans la majorité des cas, ces 44 blocs sont placés de manière séquentielle au sein du même segment. La lecture de cette séquence d'enregistrements référencés par un bloc de l'index se résume à la lecture séquentielle d'un segment, soit :

**$t_{lss1} = t_{la1} + (N_b - 1) \times t_{tr} = t_{la1} + 43 \times t_{tr}$**

Cette formule donne le temps d'accès pour une série d'extensions contigües. Etant donné un chargement massif de données, ce calcul correspond parfaitement à l'exemple. Cependant, lors d'une utilisation partagée (cas le plus fréquent), les extensions ont moins de chance d'être contigües. Dès lors, la formule d'une lecture séquentielle d'extensions non contigües est plus adaptée, soit :

**$t_{lsbl} = N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$**

Le *CF* lié à un index est calculé lors de la prise de statistique. Dans cet exemple, les statistiques sont calculées à chaque validation de transaction, soit toutes les 10 000 opérations réalisées.

**Ajout ordonné décroissant**

Le second cas analysé est celui d'une table qui comporte une série d'insertions ordonnées mais avec une valeur de clés décroissantes. Dans ce cas, la technique utilisée pour l'index consiste en une série de divisions de type 50/50 sur le premier bloc de la table. Le taux d'utilisation de l'index est de 50% comme le montre la Figure 6-3 : Taux de remplissage de l'index (insertions ordonnées décroissantes HOT). L'utilisation de ce type de division entraîne que le taux de rupture de l'index est fort élevé dans ce mode d'insertion.

De par ces deux phénomènes, d'une part, le faible taux de remplissage et, d'autre part, le haut taux de rupture, le parcours séquentiel de l'index n'est pas optimum et voit son temps de lecture augmenter significativement.

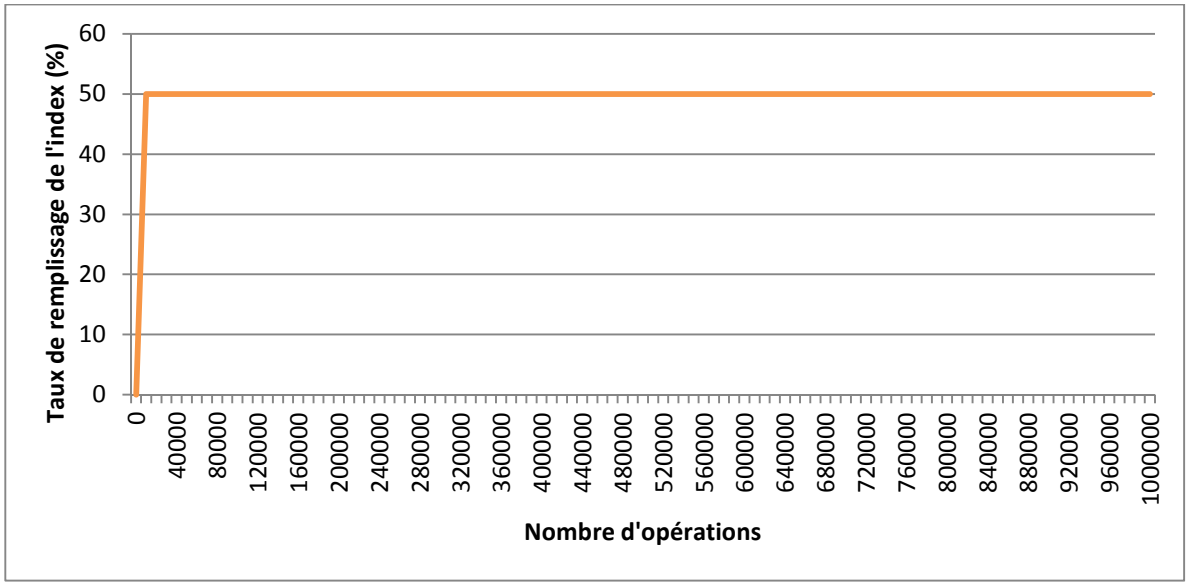


**Figure 6-3 : Taux de remplissage de l'index (insertions ordonnées décroissantes HOT)**

L'autre caractéristique intéressante concerne l'ordre des lignes dans la table. Le *CF* est idéal, cette caractéristique de la table est liée au parfait groupement des lignes au sein des blocs. En effet, lors d'une lecture séquentielle, le moteur Oracle parcourt l'index afin de trouver les lignes correspondantes à la requête. Avec les *rowID* de ces lignes, le *SGBD* connait les blocs de la table qui contiennent les enregistrements qui correspondent à la requête. Etant donné que le chargement s'est fait de manière séquentielle (décroissant), plusieurs enregistrements qui se suivent de manière logique sont placés les uns à côté des autres au sein du même bloc de la table mais de manière décroissante. Les *rowID* contenus dans la table d'index référencent, pour les clés d'index de ce groupe d'enregistrement, le même bloc de la table. Par conséquent, Oracle réalise une seule lecture pour obtenir l'ensemble des enregistrements contenus au sein d'un bloc de la table. Le même phénomène se reproduit pour chaque ensemble d'enregistrements qui est stocké dans le même emplacement lors d'une lecture groupée. Par conséquent, Oracle n'obtient qu'une fois chaque bloc lors d'une lecture séquentielle, les enregistrements suivants sont lus depuis la mémoire tampon.

La Figure 6-4 : Observation du *Clustering Factor* (*insertions* ordonnées décroissantes HOT) montre que le nombre de L/E lors de la lecture séquentielle de la table, en utilisant l'index, est équivalent au nombre de L/E lors de la lecture d'une table contenant des enregistrements ordonnés et croissants.

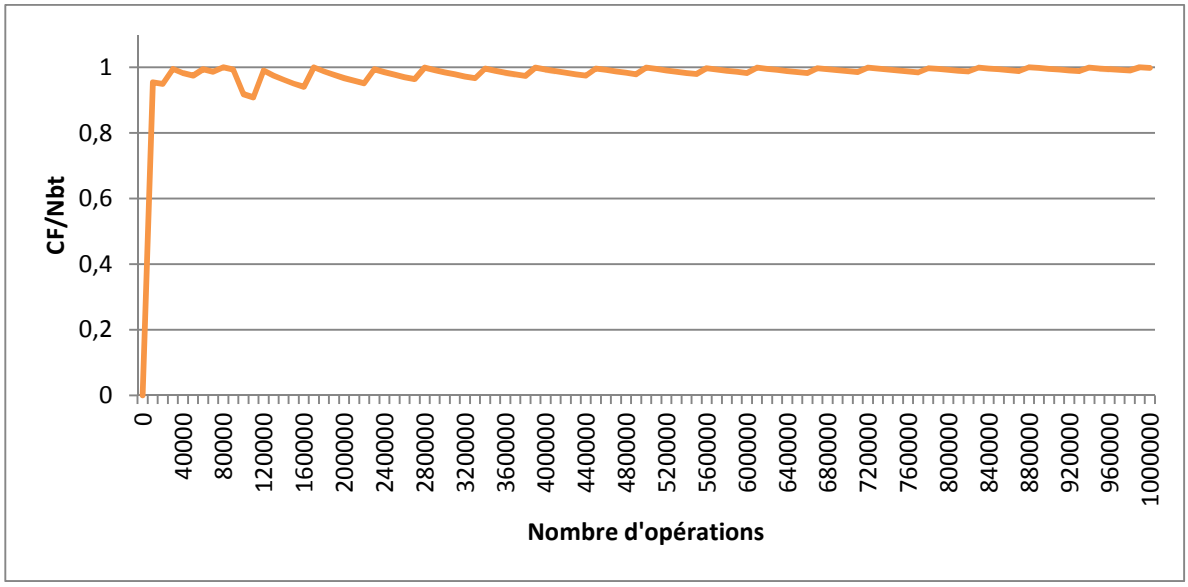


**Figure 6-4 : Observation du *Clustering Factor* (insertions ordonnées décroissantes HOT)**

Au terme de cette simulation d'ajouts ordonnés décroissants, les statistiques d'Oracle fournissent plusieurs valeurs :

**$N_{bt}$ = 83537 ; CF = 83352 ; $N_{bi}$ =** 4736 **; $N_{bf}$ = 4583 ; $N_{bb}$ = 17 ; $\tau_i$ = 0,50 ; $L_i$ = 18,2899.**

Remarque : Le nombre de blocs de la table et le nombre de blocs de l'index sont le nombre blocs alloués avec des extensions, cela ne correspond pas au nombre de blocs utilisés.

Tel qu'établi dans le modèle précédent, il est possible de calculer ces valeurs à partir du modèle établi préalablement. Celui-ci établit exactement les mêmes valeurs pour la table *HOT*

**$N_e$ = 1000000**

**$M_{epbHOT} = \lfloor L_b / L_e \rfloor$ = 15 enregistrements par bloc**

**$N_{epbHOT} = \tau_b \times \lfloor L_b / L_e \rfloor = 0.8 \times \lfloor 8000 / 511 \rfloor$ = 12 enregistrements par bloc**

**$N_{bt} = \lceil N_e / N_{epbHOT} \rceil$ = 83334 blocs de la table *HOT***

La table occupe 83334 blocs. Ce qui correspond à la taille du CF, soit le nombre de blocs à lire pour parcourir la table de manière ordonnée sur l'index.

**$M_{ipb} = \lfloor L_b / L_i \rfloor$ = 437 clés par bloc**

**$N_{ipb} = \tau_i \times \lfloor L_b / L_i \rfloor$ = 218,7 clés par bloc**

**$N_{bi} \approx N_{bi}(n) = \lceil N_e / N_{ipb} \rceil$ = 4572 blocs de l'index**

**Ajout désordonné**

Le troisième cas étudié est celui d'une table chargée avec des insertions aléatoires. Les enregistrements sont toujours insérés aléatoirement dans le segment dédié pour la table. L'organisation de l'index permet d'y accéder de manière ordonnée. La conséquence directe de ce type de chargement est que l'index subira un plus grand nombre de divisions 50/50 qui, d'une part, sont plus coûteuses que les divisions 90/10 et d'autre part, augmentent le taux de rupture de séquence (cf. section 3.3.2 ). Au final, le taux de remplissage de la table reste maximal, alors que le taux de remplissage de l'index oscille entre 65% et 75% en tendant vers 70%. Le graphique ci-dessous illustre l'évolution de l'index lors de l'insertion des lignes.

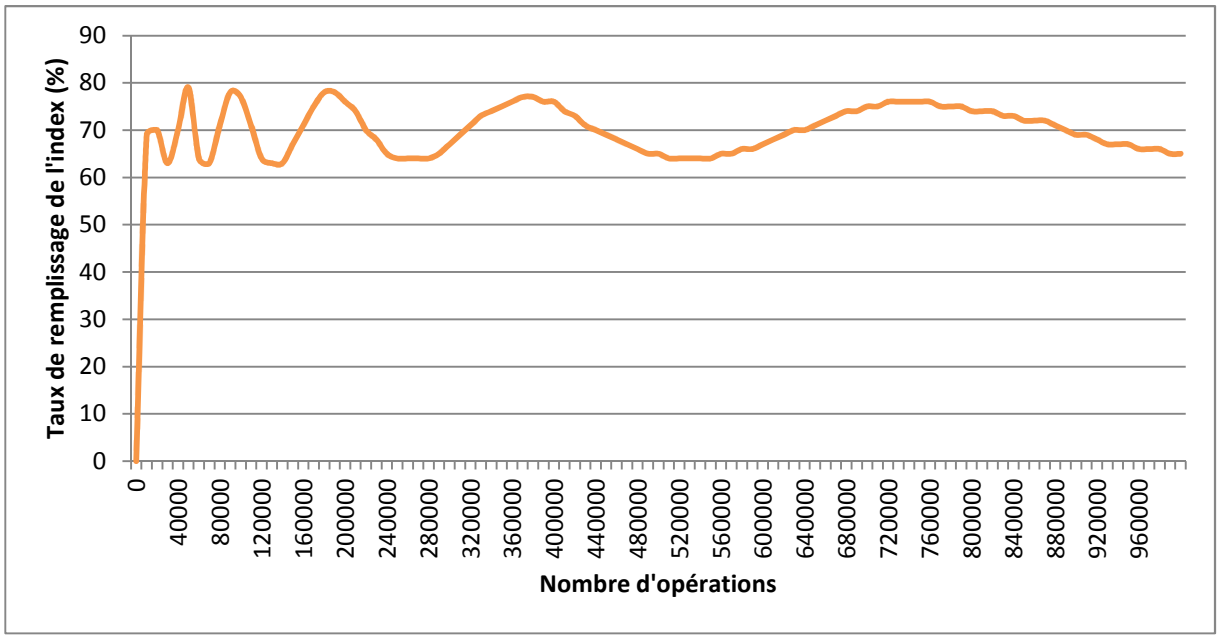


**Figure 6-5 : Taux de remplissage de l'index (insertions désordonnées HOT)**

Le comportement de cette courbe illustre un phénomène décrit dans l'annexe H de l'ouvrage « Bases de données » [3]. Lors du début du chargement, les blocs sont rapidement à leur capacité maximale. Dès lors, un grand nombre de divisions sont réalisées. Après chaque division, le taux d'occupation de l'index a tendance à diminuer. Ceci permet d'avoir de l'espace de stockage qui permet d'ajouter de nouvelles clés sans devoir réaliser de division. L'index se remplit à nouveau, jusqu'à ce qu'il ait besoin de plus d'espace. Un nouveau cycle comportant plus de divisions est réalisé. Plus la taille de l'index grandit, plus la fréquence d'oscillations de la courbe diminue. En effet, chaque phénomène de division aura un moins grand impact proportionnellement au nombre de blocs qui constituent l'index. Plus le nombre de divisions 50/50 augmentent, plus le nombre de ruptures de séquences grandit également. Les oscillations lors du début du chargement ne sont pas très significatives car elles sont liées à une procédure spécifique (de chargement) et elles sont accentuées par la faible taille de l'index.

Lors de l'insertion de ces lignes, les taux de divisions au sein de l'index sont relativement faibles. Ils sont respectivement de $\tau_{50}$ **= 0,31%** et $\tau_{90}$ **= 0,01%.**

Le second élément intéressant au niveau de l'index est son *CF*. En effet, il explose : le rapport entre le *CF* et le nombre d'enregistrements dans la table tend vers 1, le cas le plus coûteux lors des lectures séquentielles de la table, soit :

$$\frac{CF}{Ne} = \frac{999988}{1000000} \approx 1$$

Le graphique ci-dessous montre que le rapport entre le *CF* et les blocs de la table s'est nettement éloigné de 1, la valeur idéale.

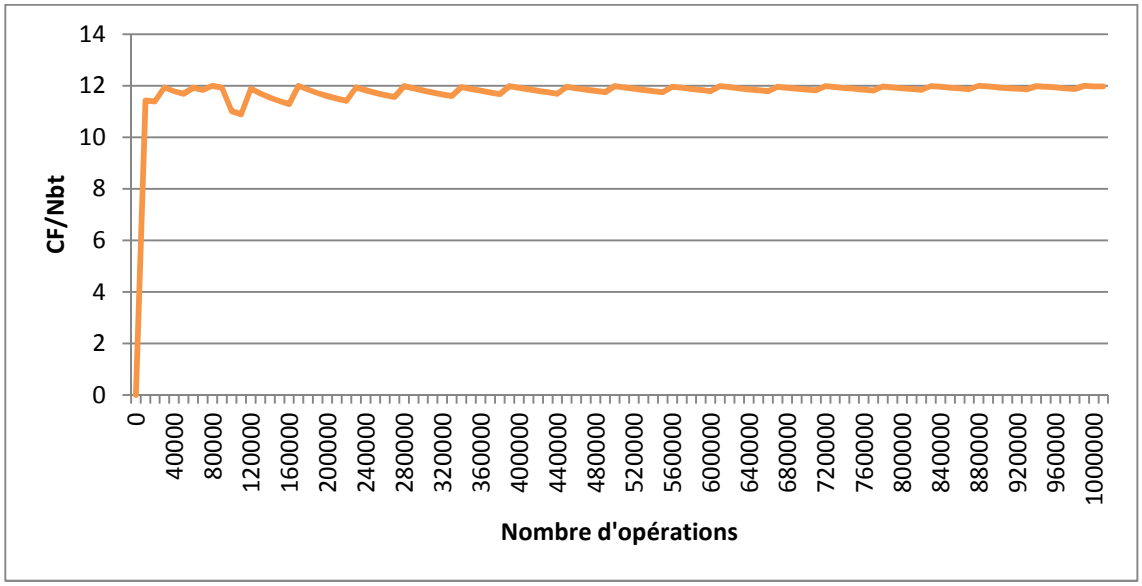


**Figure 6-6 : Observation du *Clustering Factor* (insertions désordonnées HOT)**

Au terme de cette simulation sur des insertions désordonnées, les statistiques d'Oracle fournissent plusieurs valeurs :

**$N_{bt}$ = 83537 ; CF = 999993; $N_{bi}$ = 3456 ; $N_{bf}$ = 3346 ; $N_{bb}$ = 9 ; $\tau_i$ = 0,68 ; $L_i$ = 18,1797.**

Tel qu'établi dans le modèle précédent, la table *HOT* travaillant avec les mêmes hypothèses, affiche une taille identique. Par contre, le CF est différent, il est très élevé comparativement à une situation d'ajout d'enregistrements ordonnés. La lecture séquentielle de la table a un haut taux de rupture au niveau physique.

**$N_e$ = 1000000**

**$N_{bt} = \lceil N_e / N_{epbHOT} \rceil$ = 83334 blocs de la table *HOT***

La table occupe 83334 blocs. Ce qui correspond à la taille du CF, soit le nombre de blocs à lire pour parcourir la table de manière ordonnée sur l'index.

**$M_{ipb} = \lfloor L_b / L_i \rfloor$ = 440 clés par bloc**

**$N_{ipb} = \tau_i \times \lfloor L_b / L_i \rfloor$ = 299,2 clés par bloc**

**$N_{bi} \approx N_{bi}(n) = \lceil N_e / N_{ipb} \rceil$ = 3343 blocs de l'index**

Dans cet exemple, chaque bloc de l'index référence, en moyenne, des enregistrements contenus dans 289 blocs de la table :

**CF / $N_{bi}$ = 289 blocs**

Il possible est de connaître l'indice de perte de performance en utilisant la formule définie dans la section 3.3.1.2 ). Cette formule montre le nombre additionnel de blocs de la table qu'il faut parcourir lors du parcours des clés d'index contenues au sein d'un bloc de l'index.

**Indice de perte de performance** = $\frac{|\ CF - Nbt\ |}{Nbt} \approx 11$

La différence majeure par rapport au point précédent (ajout ordonné), est la non contigüité des blocs lors des divisions de type 50/50. Le scénario de lecture correspond donc à une lecture sur un disque fortement partagé, sans lecture anticipée, soit :

$$t_{lss3} = N_b \times t_{la1} = 289 \times t_{la1}$$

Dès lors, pour le parcours du contenu d'un bloc de l'index, la performance attendue est presque 230 fois moins bonne que le parcours de la table ordonnée.

### 6.6.3 Ajout et suppression

Le quatrième cas est l'étude d'une table vivante qui subit des opérations de divers types. Comme décrit à la section 3.3.2 , les mises à jour sont un cas particulier qui se résume à une suppression suivie d'une insertion. Dans l'exemple, les mêmes tests que ceux présentés précédemment sont réalisés en modifiant le taux d'insertion qui variera entre 60% et 100%. Ces opérations sont réalisées de la même manière qu'auparavant, de façon ordonnée et désordonnée. Dans un premier temps, il s'agit d'ajouts d'enregistrements dont la clé d'index est strictement croissante avec un pourcentage de suppressions. Ensuite, le même type d'opérations est réalisé, mais en effectuant des ajouts désordonnés.

**Ajout ordonné et suppression désordonnée**

Lors des ajouts de données ordonnées, force est de constater que le taux de remplissage de l'index tend rapidement à se stabiliser vers une asymptote horizontale en fonction du taux de suppression que subit la table et son index. L'espace disponible et, par conséquent, la perte de performance liée à l'espace non utilisé au sein de la table sont dus au mode d'insertion et de suppression. Lors de chaque opération d'ajout, une opération d'insertion sur le dernier bloc (ajout ordonné) est réalisée. Le dernier bloc est donc divisé avec la méthode 90/10, et un nouveau bloc est alloué à l'index. L'opération de suppression est réalisée au sein de chaque bloc de manière aléatoire et aucun bloc ne se retrouve libéré. En cas de fort taux de suppression, cette procédure induit deux conséquences:

- la diminution du taux de remplissage de l'index ;
- une densité variable des blocs au sein de l'index. Les blocs de l'index sont peu remplis dans les valeurs inférieures et sont fortement remplis dans les valeurs supérieures.

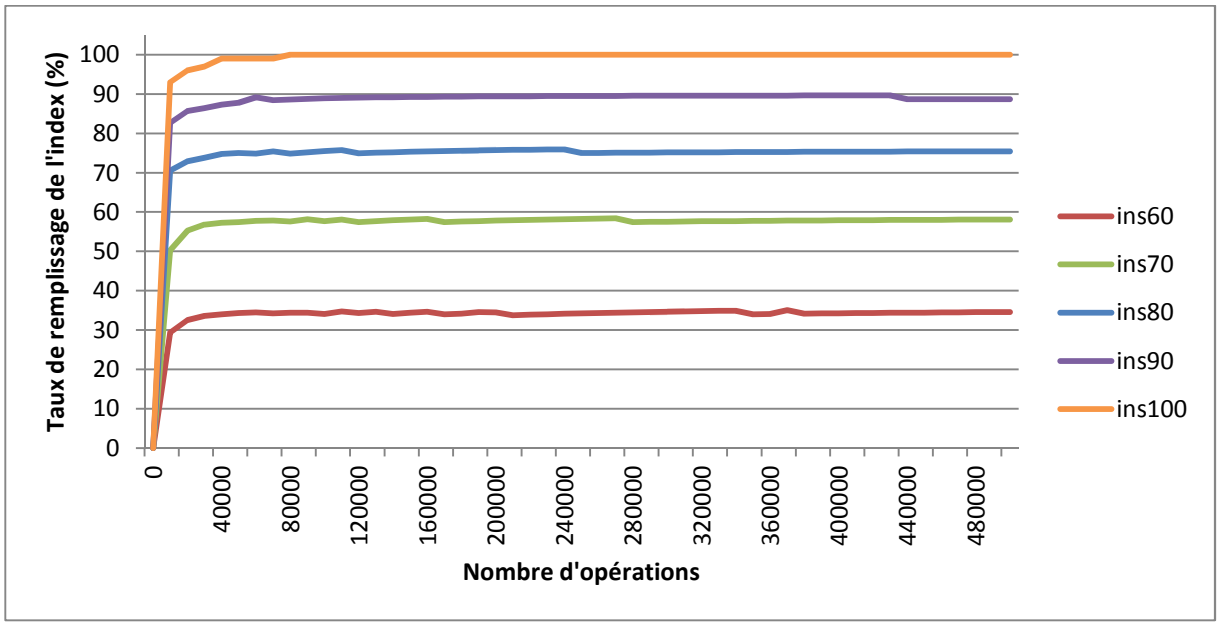


**Figure 6-7 : Taux de remplissage de l'index lors d'opérations d'insertions ordonnées et de suppressions (HOT)**

Les performances de l'index dans cette configuration diminuent en fonction du taux de suppression appliqué. Le parcours de l'index est d'autant plus optimisé qu'il est compact. D'après cette projection, il est probable qu'il faille analyser l'état de l'index à partir de 10% de suppressions sur les opérations de type *DML*. Au-delà de 20% de suppressions dans les opérations de *DML*, la perte de performances s'accentue davantage.

Le graphique ci-dessous permet d'observer l'impact de ces opérations sur l'organisation des enregistrements dans la table. Ces derniers sont parfaitement ordonnés lors d'une série d'insertions sans suppression (ins100). La lecture séquentielle des enregistrements de la table voit ses performances se dégrader en fonction du facteur de suppressions. Les performances sont diminuées d'un facteur respectif de 1,25 (ins90); 2,73 (ins80); 4,45 (ins70) et 5,99 (ins60). Ces facteurs représentent le coût en L/E supplémentaires pour lire séquentiellement **n** enregistrements dans chacune des situations.

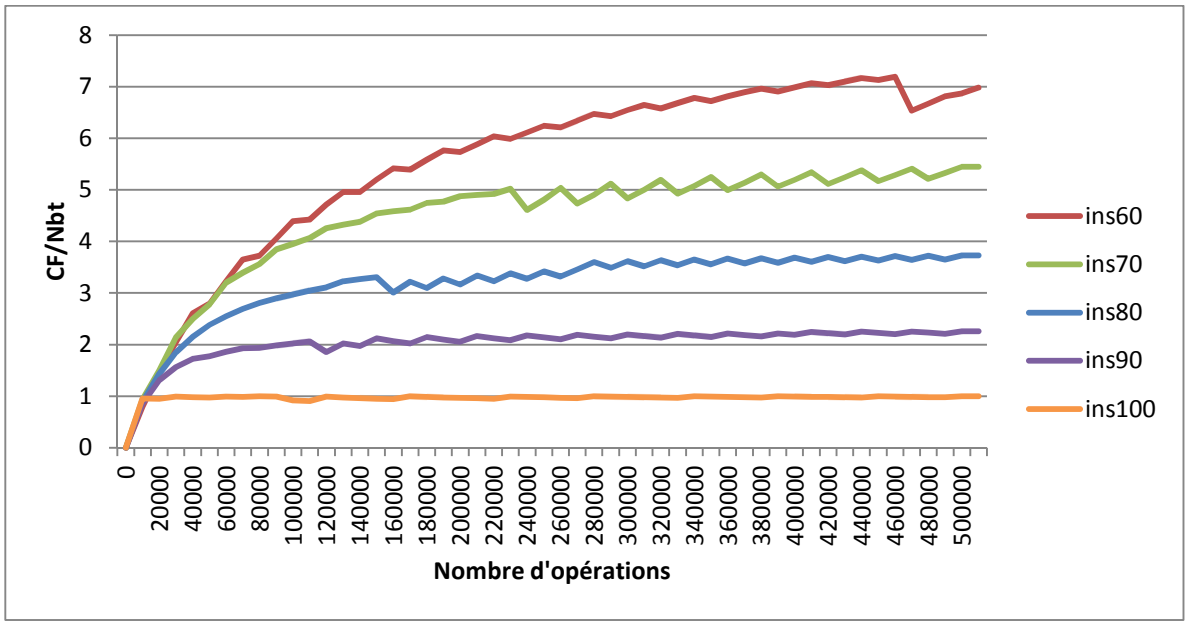


**Figure 6-8 : Etat du *CF* en fonction de différents pourcentages d'insertions ordonnées (HOT)**

### Ajout désordonné et suppression désordonnée

Le comportement du taux de remplissage de l'index lors d'ajouts et de suppressions désordonnés se distingue nettement du cas précédent (cf. Figure 6-9 : Taux de remplissage, erroné, fournit par les statistiques de l'index lors d'opérations d'insertions désordonnées et de suppressions). Quel que soit le taux d'insertion, le taux de remplissage de l'index oscille entre 65 et 75% en tendant également vers 70%. La proportion de suppressions influence uniquement la fréquence du cycle. Cette différence de comportement par rapport aux tests précédents est due à la réutilisation de l'espace supprimé au sein des blocs. Chaque insertion peut engendrer trois résultats principaux possibles:

- L'insertion est réalisée dans un bloc avec de l'espace disponible: c'est le cas le plus économe, la valeur est simplement ajoutée;
- L'insertion est réalisée dans un bloc qui a subi des suppressions: le bloc est nettoyé des valeurs effacées et la ligne y est insérée;
- L'insertion est réalisée dans un bloc plein: le bloc doit subir une division de type 50/50.

La proportion de ces trois cas varie principalement en fonction de la valeur de $\mathbf{M_{ipb}}$ et du taux de suppression. Lorsque le taux de suppression est élevé, la proportion de divisions de type 50/50 diminue car l'espace est réutilisé. Ce phénomène se traduit par une diminution du taux d'oscillations des courbes qui représentent le taux de remplissage de l'index comme le montre la Figure 6-9.

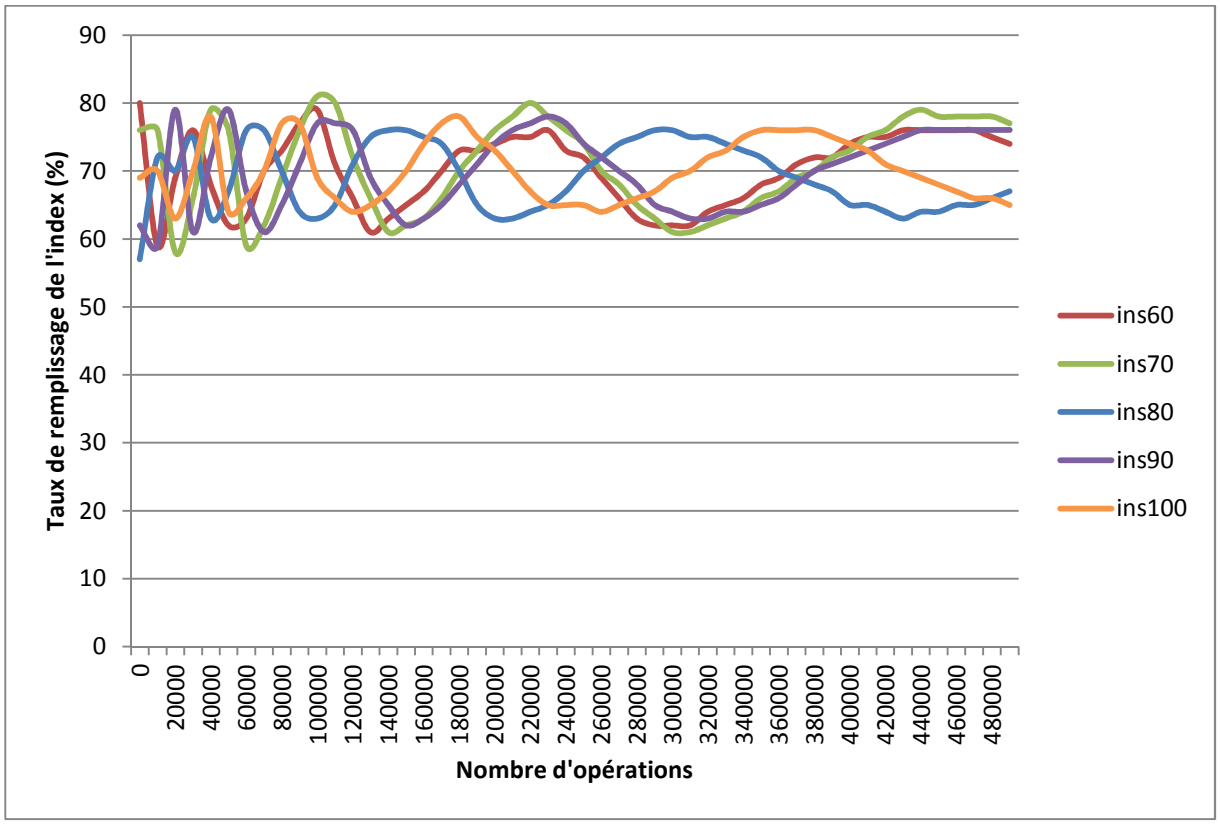


**Figure 6-9 : Taux de remplissage, erroné, fournit par les statistiques de l'index lors d'opérations d'insertions désordonnées et de suppressions (HOT)**

Le taux de remplissage de l'index reste relativement stable quel que soit le taux d'insertion. Ce comportement est anormal. En théorie, si le taux de suppression est plus élevé, le taux de remplissage devrait diminuer. En analysant les statistiques, il apparaît que le taux de remplissage reste identique même après une suppression massive. Oracle ne supprime pas les lignes, il se contente de ne pas les recopier lorsqu'il y a une modification dans le bloc. Dès lors, les statistiques sur le taux de remplissage de l'index sont faussées lors d'opérations de suppressions.

Oracle fournit, dans les statistiques d'un index, la notion de lignes supprimées qui n'ont pas encore été nettoyées. La différence entre le nombre de lignes dans la table et le nombre de lignes supprimées indique le nombre de lignes qui doivent réellement être prise en compte au sein de la table. Ceci permet d'obtenir le taux d'utilisation réel de l'index. La figure ci-dessous montre le taux de remplissage réel des index en fonction des différentes situations.

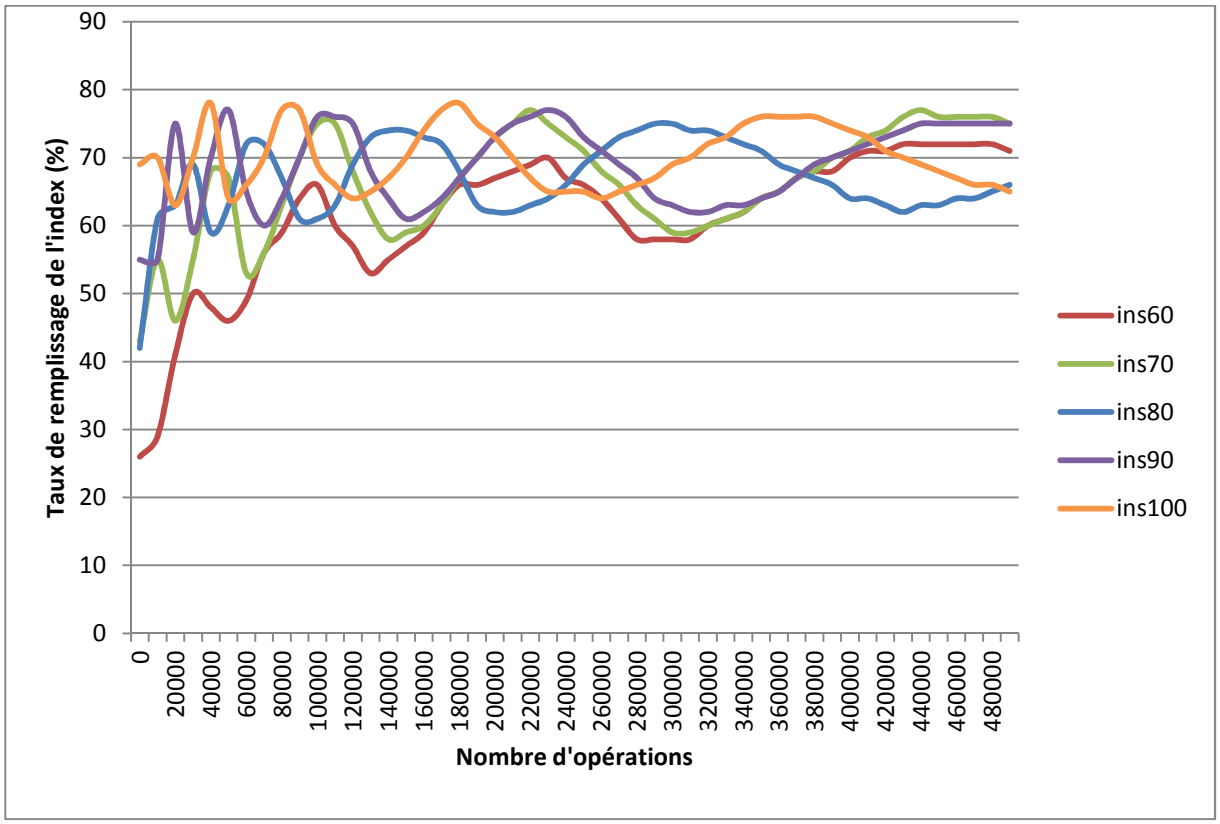


**Figure 6-10 : Taux de remplissage réel de l'index lors d'opérations d'insertions désordonnées et de suppressions (HOT)**

Le taux de remplissage dans ce mode tend également vers 70%. Au départ, le taux de remplissage est faible. Mais à mesure que l'index grandit, son taux de remplissage se stabilise et l'espace supprimé est réalloué à une autre clé d'index.

Le *CF* est très mauvais. Cela n'aurait pas pu être différent en raison de la désorganisation des enregistrements au sein de la table *HOT* due aux insertions aléatoires. Le taux de rupture lors de l'accès à cette table est très mauvais.

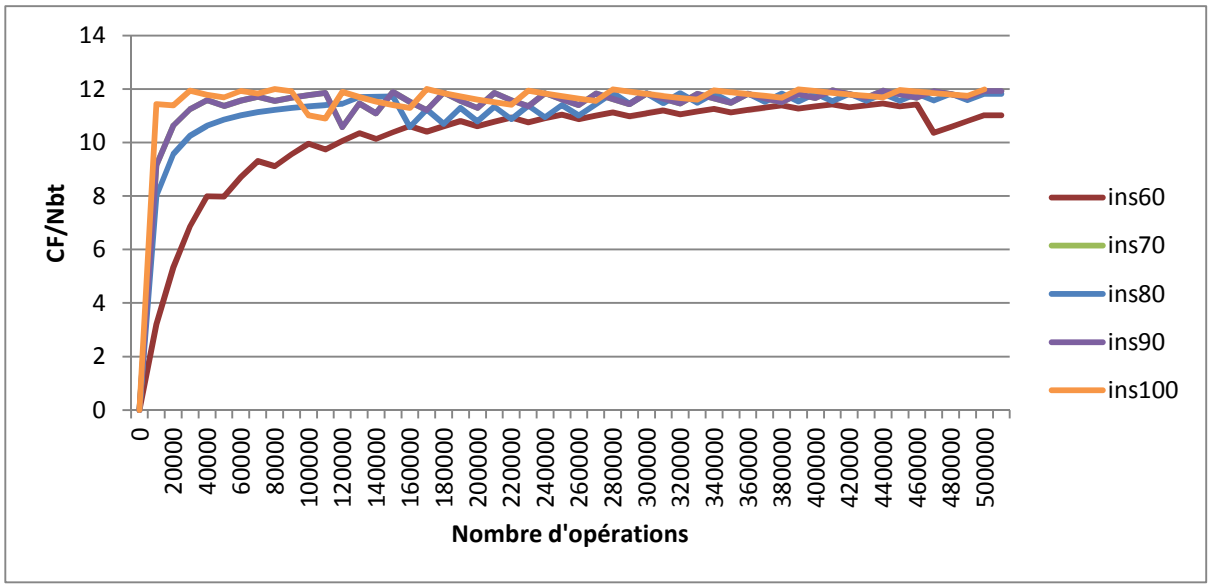


**Figure 6-11: Etat du *CF* lors d'insertions désordonnées et de suppressions (HOT)**

## 6.7 Synthèse

Cette synthèse clôture l'analyse de la structure *HOT* et de son index associé. La première partie présente la manière de calculer le volume d'une table *HOT* ainsi que son index associé dans le but de quantifier le nombre de blocs liés aux opérations sur la table. Ensuite, le temps de lecture des différents modes d'accès est modélisé pour cette structure (la lecture de la table, la lecture d'un enregistrement par l'index et la lecture d'une séquence d'enregistrements par l'index).

Logiquement, suit la définition du coût des différentes opérations réalisables sur ces structures: insertions, suppressions et modifications. Les insertions introduisent des opérations de divisions (opérations très coûteuses en ressources), permettant ainsi de modéliser les deux types de divisions et connaître leur impact respectif sur les performances. Ensuite, les conséquences liées aux taux de remplissage de l'index et la problématique du CF sont abordé afin de montrer les performances à la limite des modèles. Ceci permet de comprendre l'importance de l'ordre des enregistrements au sein de la table *HOT*. Enfin, plusieurs simulations sont effectuées sur une table *HOT* et son index afin d'analyser l'impact sur les performances de cette dernière.

# 7 Analyse d'une table *IOT*

## 7.1 Préambule

Précédemment, l'étude de la Table *HOT* et de son index a été exposée. Il reste maintenant à réaliser la même étude pour la troisième structure analysée dans ce mémoire.

Pour une structure de type *IOT*, quelles sont les métriques possibles à calculer et comment se comporte cette structure dans les différents scénarios présentés dans la section 5?

Quel est le temps d'accès nécessaire à un enregistrement ou à une séquence d'enregistrements? Quels sont les temps nécessaires à la réalisation des différentes opérations DML et quel impact sur les performances peut induire le taux de remplissage de la table?

### 7.1.1 Volume d'une table *IOT*

Lors de la création d'une table *IOT*, deux structures sont créées sur le disque : une table et un index. La table est un containeur qui ne reçoit aucune donnée. Une structure de type index est également créée. C'est elle qui stocke les enregistrements dans ce type de structure. Par conséquent, le volume d'une table *IOT* comprend deux segments générés : Table + Index. Le volume de la table est nul et le reste. C'est la taille du segment de type index qui croit avec les données. Par souci de compréhension, ce document continue à nommer cette structure « table » même si les informations sont stockées au sein d'un segment de type index.

Le segment de type index est composé de deux type de bloc : les blocs « branches » (incluant le cas particulier de la racine) et les blocs « feuilles ».

La taille d'une table *IOT* est la somme du nombre de blocs « branche » ($\mathbf{N_{bb}}$) et du nombre de blocs « feuille » ($\mathbf{N_{bf}}$). Elle est notée:

$$\mathbf{N_{b\mathit{IOT}} = N_{bb} + N_{bf}}$$

Le nombre de blocs « branche » est lié au nombre de blocs « feuille », à la taille de la clé d'index et à la taille allouée aux données dans un bloc pour stocker ces lignes, soit :

$$\mathbf{N_{bb} = N_{bf} \times L_c / L_b}$$

Sous certaines conditions, cette formule peut être simplifiée. En effet, les blocs « branche » référencent une valeur de clé dont la taille est souvent fortement inférieure à la taille du bloc. Par conséquent, le rapport de la taille de la clé par rapport à la taille du bloc est très faible, voire négligeable.

Dans cette situation, la taille d'une table *IOT* peut être associée au nombre de blocs « feuilles » qui la composent, soit :

$$\mathbf{N_{b\mathit{IOT}} \approx N_{bf}}$$

Il est important de mettre en avant une différence par rapport à la table *HOT*: un index réserve de la place pour des ajouts lors de sa reconstruction, mais ne conserve pas de place pour les mises à jour au sein de ses blocs. La table *HOT* conserve un espace dédié aux mises à jour (paramètre PCTU*SE*D soit $\tau_{b\mathit{HOT}}$). Après un chargement identique contenant uniquement des ajouts ordonnés, une table

*IOT* utilise 100% de l'espace qui lui est alloué (N.B. pour rappel, lors d'insertions ordonnées, des divisions de type 90/10 sont réalisées. La section 3.4.2 rappelle qu'il s'agit d'un abus de langage et explique que ce type de division sont en réalité des divisions 99/1), ce qui n'est pas le cas d'une table *HOT*. Par conséquent, le calcul de taille d'une table *IOT* peut s'écrire tel que :

$$N_{bIOT} = N_{bHOT} \times \tau_{HOT}$$

En généralisant, on peut noter que le nombre de blocs qui composent une table *IOT,* multiplié par son taux de remplissage moyen à un moment donné, équivaut au nombre de blocs qui compose une table *HOT* contenant les mêmes valeurs également multiplié par son taux moyen de remplissage, soit :

$$N_{bIOT} \times \tau_{IOT} = N_{bHOT} \times \tau_{HOT}$$

où $N_{bHOT}$ représente le nombre de blocs contenant les mêmes données que dans une table *HOT* et où, à un moment donné, $\tau_{HOT}$ et $\tau_{IOT}$ sont respectivement les taux de remplissage de la table *HOT* et de la table *IOT*.

**Calcul de la taille d'un table $N_{bIOT}$**

La table contient $N_e$ enregistrements de taille $L_e$ stockés dans les $N_{bIOT}$ blocs alloués à la table. Ces blocs ont une taille $L_b$ qui est fixe. $\tau_{bIOT}$ qui est le taux de remplissage moyen des blocs de la table à un moment donné, est également connu. A partir de ces notions, il est possible d'obtenir $M_{epbIOT}$ et $N_{epbIOT}$ qui sont respectivement le nombre maximum et le nombre moyens d'enregistrements par bloc « feuille ».

$$M_{epbIOT} = \lfloor L_b / L_e \rfloor$$

$$N_{epbIOT} = \tau_{bIOT} \times M_{epbIOT}$$

Pour une table qui se remplit avec des ajouts séquentiels croissants ou avec des ajouts séquentiels décroissants, le $\tau_{bIOT}$ est de 1 (grâce à son mécanisme de division voir 3.4.2 ). Par conséquent, le nombre moyens d'enregistrements par bloc « feuille » est :

$$N_{epbIOT} = \lfloor M_{epbIOT} \rfloor$$

A partir de ces éléments, le calcul du nombre de blocs « feuille » dans une table est :

$$N_{bIOT} = \lceil N_e / N_{epbIOT} \rceil$$

## 7.2 Temps de lecture

Le temps de lecture d'une table *IOT* se résume au temps de parcours de l'index. Dès lors, le temps de lecture est celui qui est nécessaire au parcours d'un objet de type index.

### 7.2.1 Temps de lecture d'un enregistrement

Au niveau physique, il s'agit de $N_{bla}$ lectures aléatoires : soit la hauteur de l'index (**H**) représentée tel que :

$$N_{bla} = H$$

Le temps de lecture d'un enregistrement se résume au parcours de l'index, soit:

$$T_{lc} = N_{bla} \times t_{la1}$$

### 7.2.2 Temps de lecture d'une séquence d'enregistrements

Lorsqu'il faut accéder, en lecture, à une séquence d'enregistrements, la structure de type *IOT* est très efficiente. En effet, les blocs « feuille » sont doublement chaînés entre eux. Par conséquent, il suffit d'atteindre la première valeur **$(H \times t_{la1})$** puis de lire de manière séquentielle les blocs suivants. Le temps de lecture séquentielle d'un segment a déjà été calculé à la section 5.4 , il s'agit de :

$$t_{lsbl} = N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$$

Rappel : La section 5.4 définit $N_{Ec}$ comme étant le nombre de groupe d'extensions contigües sur le disque. Une extension seule incrémente également ce nombre.

Cette mesure fournit le temps nécessaire à la lecture d'une séquence d'enregistrements au sein d'un segment de type *IOT*. Dans le cas idéal, après un chargement ou lors d'une table utilisée uniquement avec des insertions sur des valeurs de clés croissantes. Par conséquent, son temps de lecture est :

$$t_{lsIOT} = H \times t_{la1} + N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr}$$

Par la suite, chaque phénomène de chaînage ($N_{ch}$) provoque une lecture supplémentaire lors de l'accès à un enregistrement. Dès lors le temps de lecture d'une séquence d'enregistrement devient:

$$t_{lsIOT} = H \times t_{la1} + N_{Ec} \times t_{la1} + (N_b - N_{Ec}) \times t_{tr} + N_{ch} \times t_{la1}$$

En cas de chargement des données de manière non ordonnée par rapport à la clé d'index, les blocs de la table subissent beaucoup de divisions. Lors de chaque division de type *IOT*, un bloc libre est ajouté dans la structure logique du B-tree. Ce phénomène a deux conséquences directes : d'abord, cette division augmente le taux d'espace vide au sein de la table, ensuite, cette division empêche de réaliser une lecture séquentielle des blocs du segment. En effet, lors de ce type de lecture, il faut parcourir une suite de blocs séquentiels ainsi que le bloc ajouté, par la suite, dans la structure. Ce bloc additionnel est localisé en fin de segment dans la majeure partie des situations. A mesure que ce type de division se produit sur le segment, la performance de la lecture séquentielle de celui-ci diminue.

### 7.2.3 Temps d'ajout d'un enregistrement

Lors de l'insertion, il faut accéder au bloc correspondant dans l'index ($t_{lci}$) puis réécrire ce bloc. Si le bloc ne peut fournir l'espace nécessaire, une opération de division doit être réalisée. Il existe donc trois scénarios:

- Sans division:

  Il faut réécrire le bloc « feuille » avec le nouvel enregistrement soit :

  $$t_{ins} = t_{lc} + t_{la1}$$

- Soit division 90/10:

  Ce type de division se produit en cas d'ajout d'une valeur plus grande que n'importe quelle autre valeur au sein de la table. Dans ce cas, il faut obtenir l'adresse d'un nouveau bloc. Puis, il faut écrire le nouvel enregistrement au sein de ce bloc ($t_{la1}$). Ensuite, il faut mettre à jour le

pointeur contenu dans le bloc précédent ($t_{la1}$) pour enfin mettre à jour le bloc « branche » correspondant ($t_{la1}$).

$$t_{ins90} = t_{lc} + 3 \times t_{la1}$$

- Soit autre division:

  Dans ce cas, il faut obtenir l'adresse d'un bloc vide. Il faut ensuite réécrire les 2 blocs contenant les enregistrements ($2 \times t_{la1}$). Il faut également mettre à jour le bloc « branche » correspondant ($t_{la1}$) pour enfin mettre à jour le pointeur contenu dans le bloc qui suit la division, vers le nouveau bloc ($t_{la1}$).

$$t_{insxx} = t_{lc} + 4 \times t_{la1}$$

Ces deux types de divisions ont une certaine probabilité de se produire, et sont représentés de la manière suivante :

$\tau_{90}$ est la probabilité d'une division 90/10 et $\tau_{xx}$ est la probabilité d'une autre division.

Lors d'insertions de clés ordonnées et croissantes, la technique utilisée sera celle du 90/10 où $\tau_{90} = 1 / M_{epb\mathit{IOT}}$.

Lors d'insertions de clés aléatoire, la technique utilisée sera celle d'une division *IOT* où

$\tau_{xx} = 2 / M_{epb\mathit{IOT}}$ et $\tau_{90}$ est proche de 0.

C'est sur ce point que la table de type *IOT* est la moins efficiente: le $\tau_{50}$ d'un index lié à une table *HOT* et le $\tau_{xx}$ d'une table *IOT* ne sont pas équivalents. Le rapport entre $L_{e\mathit{IOT}}$ et $L_i$ fournit un coefficient qui permet d'anticiper l'augmentation de la fréquence de $\tau_{xx}$ pour une table de type *IOT*, tel que :

$$\tau_{xx} = \tau_{50} \times L_{e\mathit{IOT}} / L_i$$

### 7.2.4 Temps de suppression d'un enregistrement

Lors de la suppression d'un enregistrement, Oracle marque la ligne comme "supprimée" à l'aide d'un drapeau. Le coût de l'opération correspond à celui de l'accès au bloc « feuille » correspondant, soit $t_{la1}$. Comme cela est signalé préalablement, le *SGBD* réalise le nettoyage du bloc lors d'une insertion. Il existe un cas particulier en cas de modification du dernier enregistrement (probabilité de $1/ N_{epb\mathit{IOT}}$). Dans ce cas, il faut modifier le bloc « branche » correspondant, soit un coût additionnel de $t_{la1}$.

$$t_{sup} = t_{lc} + t_{la1}$$

### 7.2.5 Temps de modification d'un enregistrement

La modification d'un enregistrement dure $t_{la1}$. Le moteur du *SGBD* réécrit le bloc modifié. Lors de la modification d'une clé liée à un enregistrement, le *SGBD* réalise une suppression de la clé au niveau de l'index puis effectue un ajout d'une nouvelle clé au sein de l'index. Dans le meilleur des cas, si la nouvelle clé peut être placée au sein du même bloc, l'opération dure $t_{la1}$ (soit le même exemple que celui repris dans la section 6.3.4 : changement d'orthographe d'un mot: DUPONT deviendrait DUPOND). Dans le cas le moins bon, il faut noter l'enregistrement comme supprimé puis l'ajouter dans un autre bloc. L'opération de modification dure entre $t_{la1}$ et $2 \times t_{la1}$.

$$t_{lc} + t_{la1} < t_{mod} < t_{lc} + 2 \times t_{la1}$$

## 7.3 Synthèse des calculs

$N_{bIOT}=N_{bHOT} \times \tau_{HOT}$

$M_{epbIOT} = \lfloor L_b / L_e \rfloor$

$N_{epbIOT} = \lfloor \tau_{bIOT} \times M_{epb} \rfloor$

$N_{epbIOT} = \lfloor M_{epb} \rfloor$

$N_{bIOT} = \lceil N_e / N_{epb} \rceil$

Temps de lecture d'une séquence d'enregistrement

$t_{lsbl} = N_{Ec} \times t_{la1} + (b - N_{Ec}) \times t_{tr}$

Temps de lecture d'un enregistrement

$T_{lc} = N_{bla} \times t_{la1}$

## 7.4 Impact du taux de remplissage de la table *IOT*

Cette fois, le nombre de lectures et d'écritures (L/E) nécessaires afin de réaliser une requête, varie uniquement en fonction du taux de remplissage de l'index. Cette section calcule la différence dans le pire et le meilleur cas de ce taux.

La section précédente décrit déjà les deux situations extrêmes, soit un taux de remplissage de 100% ou un taux de remplissage de 50%.

Dans cet exemple, la taille des blocs est de 16Ko. Sur cet espace, seuls 16000 octets sont alloués pour le stockage des données. Le reste du bloc est réservé pour stocker les informations de l'entête (cf. section 2.2.2

Voici un exemple empirique avec une table *IOT* totalement remplie d'une part, et une table *IOT* à moitié remplie d'autre part.

Le tableau est composé de :

- 1000000 de lignes ;
- 10000 blocs de données (soit 100 lignes par bloc) ;
- Chaque enregistrement occupe un espace de 160 octets.

Les blocs constituant la table sont remplis à 100% **(A)**:

- Hauteur de l'index est de 3 ;
- Le nombre de blocs « branche » inclus dans l'index est 100 ;
- Le nombre de blocs « feuille » qui composent l'index est 10000.

Les blocs constituant la table sont remplis à 50% **(B)**:

- Hauteur de l'index est de 3 ;
- Le nombre de blocs « branche » inclus dans l'index est le double, soit 200 ;
- Le nombre de blocs « feuille » qui composent l'index est également le double soit 20000.

(BR=Bloc « racine » ; BB=Bloc « branche » ; BF=Bloc « feuille »)

Les cadres ci-dessous illustrent le calcul des coûts de différentes requêtes. Dans la section 6.5 , il a été démontré que chacune de ces requêtes a un impact différent en termes de L/E. Après chaque

cadre, le pourcentage de perte de performances dû au faible taux de remplissage de l'index est donné.

*Requête sélectionnant une ligne :*
*Cas **(A) = 1 BR + 1 BB + 1 BF = 3 L/E***
*Cas **(B) = 1 BR + 1 BB + 1 BF = 3 L/E***

Dans le cas de la sélection d'une ligne, le fait que l'index soit réparti sur un grand nombre de blocs ne change absolument pas le nombre de lectures/écritures.

*Requête sélectionnant 1000 lignes de manière séquentielle (soit 0.1% de la table) :*
*Cas **(A) = 1 BR + 1 BB + 0.001*10000 BF = 12 L/E***
*Cas **(B) = 1 BR + 1 BB + 0.001*20000 BF = 22 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de 45,46%.

*Requête sélectionnant 10000 lignes de manière séquentielle (soit 1% de la table) :*
*Cas **(A) = 1 BR + 1 BB + 0.01*10000 BF = 102 L/E***
*Cas **(B) = 1 BR + 1 BB + 0.01*20000 BF = 202 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de 49,51%.

*Requête sélectionnant 100000 lignes de manière séquentielle (soit 10% de la table) :*
*Cas **(A) = 1 BR + 1 BB + 0.1*10000 BF = 1002 L/E***
*Cas **(B) = 1 BR + 1 BB + 0.1*20000 BF = 2002 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de 49,95%.

*Requête sélectionnant 1000000 lignes de manière séquentielle (soit 100% de la table) :*
*Cas **(A) = 1 BR + 1 BB + 1*10000 BF = 10002 L/E***
*Cas **(B) = 1 BR + 1 BB + 1*20000 BF = 20002 L/E***

La perte de performance en lecture dû au taux de remplissage de l'index est de 50%

**Synthèse des exemples**

Les exemples ci-dessus permettent de mettre en avant deux observations :

- En comparant les résultats obtenus avec la table *HOT* (cf. section 6.5 il est possible de remarquer un gain de performance en lecture de la table *IOT* lorsque son taux de remplissage est élevé.
- Si la table *IOT* à un haut taux de remplissage, les performances en lecture sont maximale. Celles-ci sont meilleurs que les performances d'une table *HOT* combinée à un index avec un CF faible.

## 7.5 Phénomène de perte de performances d'une table *IOT*

La table de type *IOT* n'a pas le même fonctionnement que la *HOT*. Par conséquent, les éléments qui impactent une table *HOT*, le font différemment sur une table *IOT*. Les éléments qui affectent les performances d'une table *IOT* sont:

- sa fragmentation logique (le chaînage), chaque occurrence de ces phénomènes entraîne une lecture additionnelle ;
- le taux de remplissage de ses blocs, plus ce taux est faible, plus le nombre de blocs à parcourir pour accéder à un ensemble d'enregistrements est élevé ;
- son chargement (ordonné ou en vrac), moins les enregistrements sont ordonnés, plus les accès séquentiels entraînent des accès aléatoires dû au taux de rupture (cf. section 3.4.3 );

- son taux de modification (modifications et suppressions) plus ce taux est élevé, plus les enregistrements se désordonnent au sein de la table ;
- son taux de division.

Cette section présente les résultats de plusieurs simulations afin de permettre de voir en quoi ces facteurs influencent la performance de ce type d'objet.

### 7.5.1 Situation initiale

Le comportement d'une table de type *IOT* est analysé. Lors de sa création, la table est stockée au sein du segment qui reçoit une extension afin d'enregistrer les métadonnées lui correspondant.

Voici la structure de la table, elle est identique à celle de la table *HOT*:

| COLUMN_NAME | DATA_TYPE | NULLABLE |
|---|---|---|
| IDENT_NUM | NUMBER(20,0) | No |
| DATEI_DAT | DATE | No |
| RANDOM_VARCHAR | VARCHAR2(1000 BYTE) | Yes |

Il s'agit d'une structure assez simple comportant un chiffre identifiant, une date d'insertion et une chaîne de caractères de taille variable, initialement chargée avec 500 caractères. Cette taille initiale est choisie afin de favoriser les migrations de lignes lors des mises à jour.

Dans cette configuration, la taille des blocs est de 8Ko. Dans ceux-ci, seuls 8000 octets sont réservés pour les données, le reste du bloc étant alloués à l'entête. La taille moyenne d'une ligne au sein de la table est de 511 octets.

Remarque : La même réflexion qu'à la section 6.6.1 est à prendre en considération pour les lignes qui suivent : La taille d'un nombre stocké au sein d'Oracle est variable en fonction de sa valeur. Dans les simulations qui suivent, la taille de la valeur de la clé d'index est de 2 octets pour les valeurs de clés les plus faibles et elle est de 7 octets pour les valeurs de clés les plus élevées. Cette particularité fait que la taille de l'enregistrement et la taille de la clé d'index ne sont pas fixes. Lors des calculs de cette section, une valeur moyenne de cette taille est utilisée, ceci entraîne une légère différence entre les statistiques fournies par la simulation et le modèle défini en amont.

### 7.5.2 Table Append

#### Ajout ordonné croissant

La première étude de cas analyse l'utilisation de la table *IOT* lors d'une série d'insertions ordonnées et croissantes. L'insertion se réalise à l'aide d'une division 90/10. Cette méthode est déjà décrite précédemment.

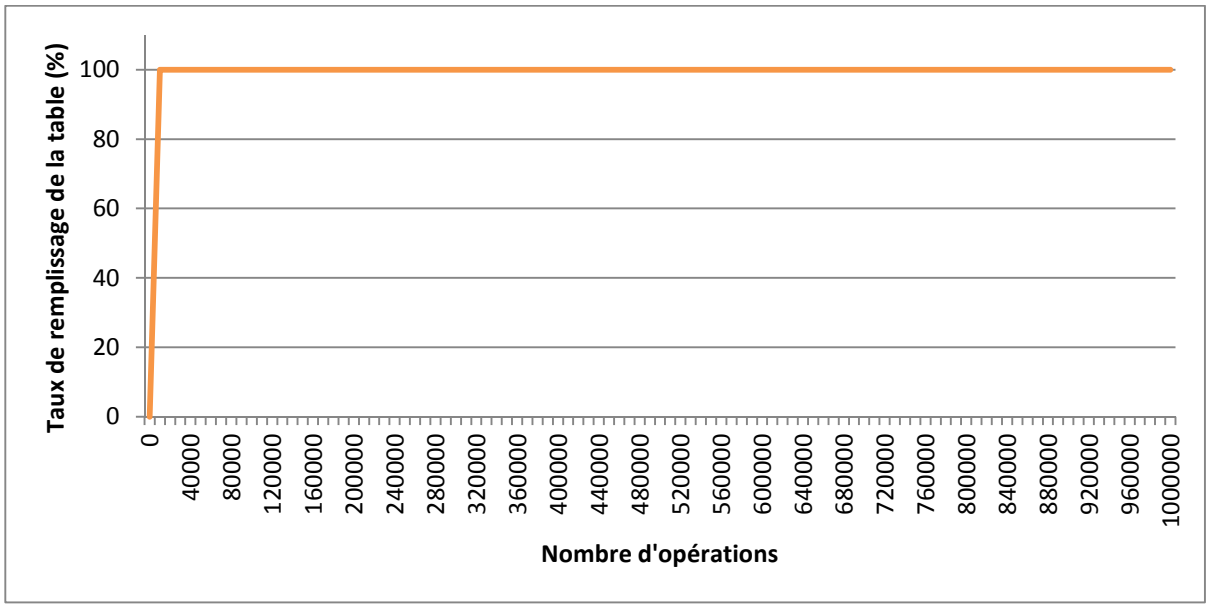


**Figure 7-1 : Taux de remplissage de l'index (insertions ordonnées croissantes)**

Ce procédé permet à Oracle de diviser uniquement le dernier bloc (division de type 90/10 cf. section 3.4.2 ), ce qui a l'avantage d'être très économique en termes de ressources. La table atteint rapidement et conserve un taux de remplissage de 100%.

Pour ce type d'insertion, la table de type *IOT* est très intéressante car le coût en L/E est plus faible. Dans ce cas, elle utilise moins d'espace et nécessite moins de ressources lors de l'exécution de *DML* sur la table.

Au terme de cette simulation sur des insertions désordonnées, les statistiques d'Oracle fournissent plusieurs valeurs :

**$N_{bIOT}$ = 67136 ; $N_{bb}$ = 3456 ; $N_{bf}$ = 66667 ; $N_{bb}$ = 93 ; $\tau_{bIOT}$ = 1 ; $L_e$ = 522,9697.**

Tel qu'établi dans le modèle précédent, la table *IOT* travaillant avec les mêmes hypothèses que celle utilisée pour la table *HOT.*

**$N_e$ = 1000000**

La table occupe **66667** blocs. Ce qui correspond à la taille du CF, soit le nombre de bloc à lire pour parcourir la table de manière ordonnée sur l'index.

**$M_{epbIOT} = \lfloor L_b / L_e \rfloor$ = 15 enregistrements par bloc**

**$N_{epbIOT} = \tau_{bIOT} \times M_{epbIOT}$ = 15 enregistrements par bloc**

**$N_{bIOT} \approx \lceil N_e / N_{epbIOT} \rceil$ = 66667 blocs de l'index**

Etant donné que les insertions sont réalisées sur des valeurs croissantes de clés, le taux d'occupation de la structure *IOT* est de 100% (cf. section 3.4.2 ). Il est possible de calculer le volume de données occupées au sein des structures *IOT* et *HOT*, soit :

$N_{bIOT} = N_{bHOT} \times \tau_{HOT}$

$N_{bHOT}$ = 66667 / 0,8

$N_{bHOT}$ = 83334 Blocs

Ceci correspond au nombre de blocs qui constituent la table *HOT* dans les mêmes hypothèses : 83334 blocs (cf. section 6.6.2 , insertions ordonnées croissantes).

**Ajout ordonné décroissant**

La seconde étude de cas analyse l'utilisation de la table *IOT* lors d'une série d'insertions désordonnées et croissantes. L'insertion se réalise à l'aide d'une division *IOT*, cette méthode est décrite précédemment dans la section 3.4.2 . Ce procédé permet à Oracle de diviser uniquement le premier bloc mais ajoute également une rupture dans la séquence physique. L'insertion est relativement économique, mais elle l'est moins qu'une division de type 90/10. L'index est atteint rapidement et conserve un taux de 100%.

Pour ce type d'insertion, la table de type *IOT* est très intéressante car le coût en L/E est relativement faible. Tout comme l'ajout ordonné croissant, elle utilise moins d'espace et nécessite moins de ressources lors de l'exécution d'opérations *DML* sur la table. Et ce, contrairement à la table *HOT* et à son index *B-tree* qui est non-efficient dans ce type de configuration.

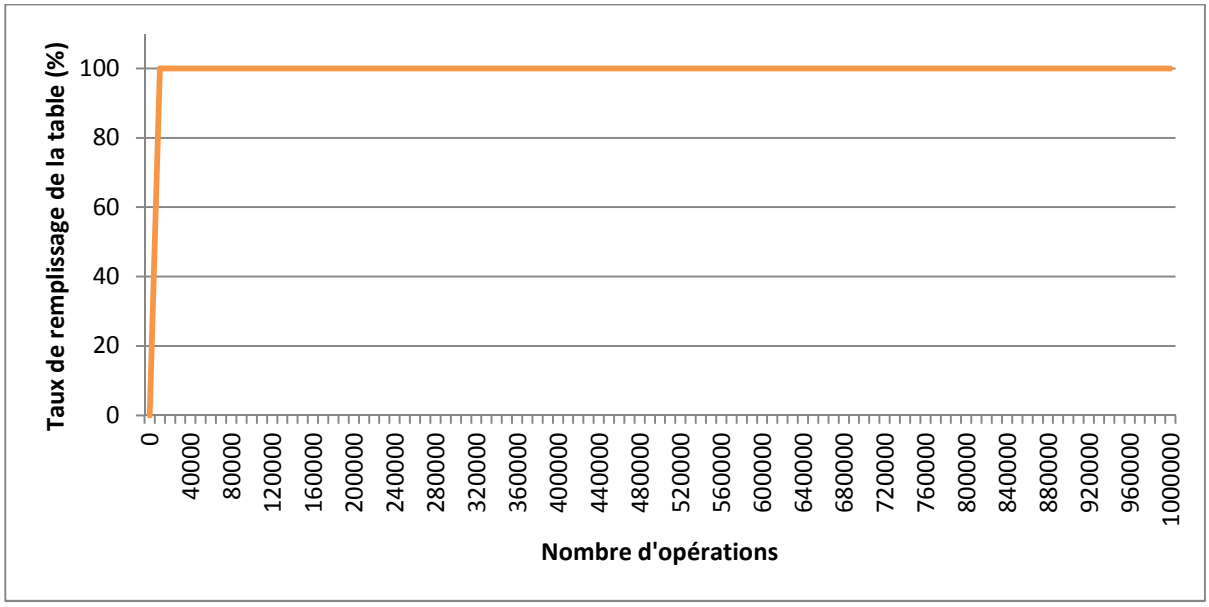


**Figure 7-2 : Taux de remplissage de l'index (insertions ordonnées décroissantes)**

**Ajout désordonné**

Le troisième cas d'étude est celui de la table chargée à l'aide d'insertions sur des valeurs de clé qui ne sont pas croissantes. Au niveau de la table de type *IOT*, un grand nombre de divisions se produit, ce qui tend rapidement à faire baisser le taux d'occupation de la table. De plus, comme le moteur Oracle garde le *B-tree* balancé, ce type de structure est fortement désavantagé par l'insertion de valeurs de clé non croissantes.

Lors du premier calcul de statistiques après 10 000 insertions, le taux d'utilisation de la table *IOT* est de 53%. Par la suite, la structure *IOT* conserve ce taux d'utilisation.

Au terme de cette simulation sur des insertions désordonnées, les statistiques d'Oracle fournissent plusieurs valeurs :

**$N_e$ = 500000 ; $N_{b/OT}$ = 63488 ; $N_{bf}$ = 62499 ; $N_{bb}$ = 129 ; $\tau_i$ = 0,53 ; $L_i$ = 525,9798.**

**$M_{epb/OT} = \lfloor L_b / L_e \rfloor$ = 15 enregistrements par bloc**

**$N_{epb/OT} = \tau_{b/OT} \times \lfloor L_b / L_e \rfloor$ = 0,53 x 15 = 7.95 enregistrements par bloc**

A partir de ces éléments, le calcul du nombre de blocs « feuille » dans une table est :

**$N_{b/OT} = \lceil N_e / N_{epb/OT} \rceil$ = 62894**

Le taux de division de type IOT est de :

**$\tau_{xx} = 2 / M_{epb/OT}$ = 13,33%**

C'est ce taux de division élevé qui entraîne que la table à un taux de remplissage relativement faible.

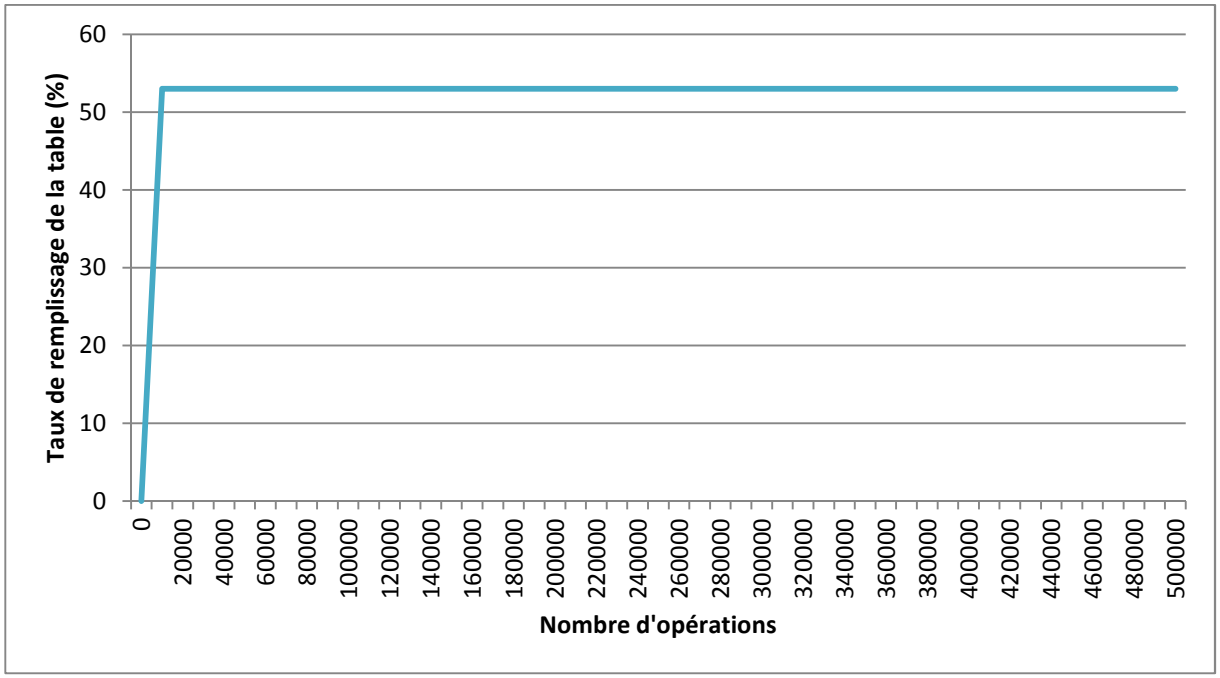


**Figure 7-3 : Taux de remplissage de l'index (insertions désordonnées)**

L'échelle du graphique ci-dessus ne permet pas d'observer le phénomène d'insertion aléatoire. Le second graphique permet de constater la stabilisation rapide vers 53% de taux d'utilisation de la table. Ce second graphique est issu du même test, mais utilise des statistiques mises à jour après chaque insertion de deux lignes. Ces données sont beaucoup plus précises. Après l'insertion de 1000 ligne, la tendance de la table se note déjà clairement.

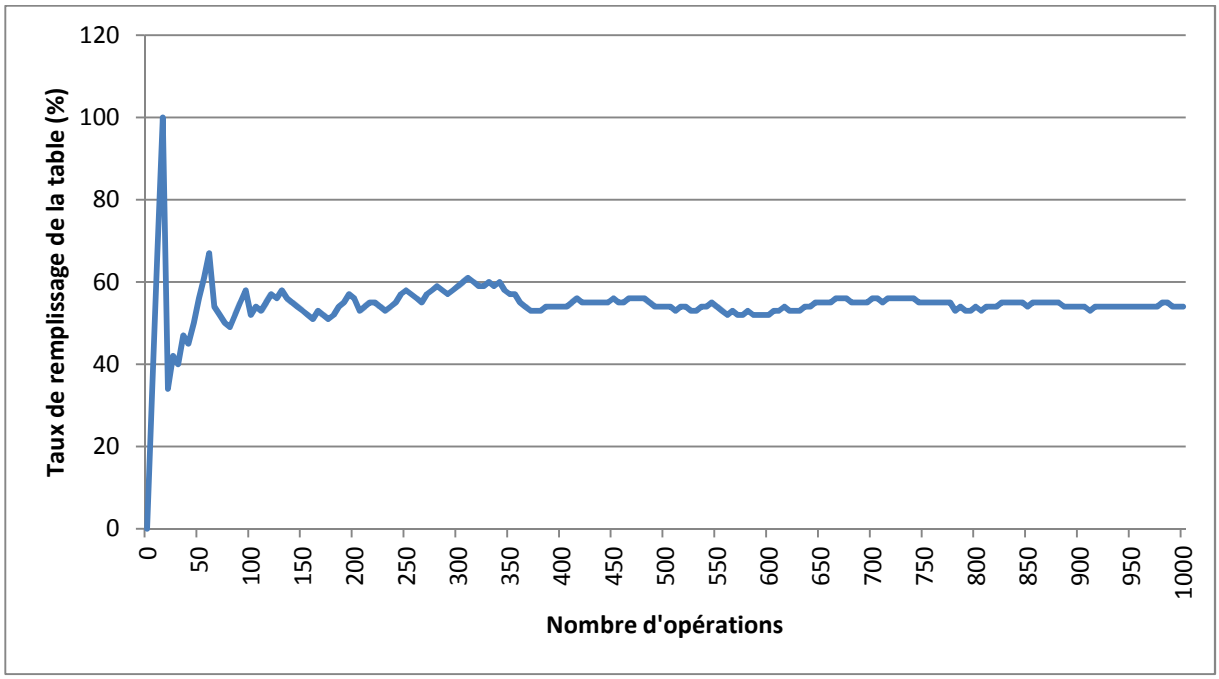


**Figure 7-4 : Taux de remplissage de l'index lors des premières insertions désordonnées**

Lors de l'insertion de ces lignes, les taux de divisions au sein de l'index est nettement plus élevé que les valeurs obtenues lors des simulations sur la structure *HOT*. Sur la table *IOT*, ils sont respectivement de $\tau_{xx}$ **= 13,12%** et $\tau_{90}$ **=0,01%.**

La perte de performance enregistrée est de presque de l'ordre de 50%.

### 7.5.3 Ajout et suppression

Le troisième cas d'étude lié aux tables *IOT* consiste en une série d'opération *DML* avec différents taux d'insertion. Comme dans l'exemple des tables *HOT*, les opérations sont des insertions et des suppressions avec un taux d'insertion qui varie entre 60% et 100%. Dans un premier temps, les insertions sont réalisées de manière ordonnée, puis désordonnée.

**Ajouts ordonnés et suppressions désordonnées**

De la même manière que pour l'index de la table *HOT*, l'ajout de données ordonnées amène le taux de remplissage de la table à rapidement se stabiliser vers une asymptote horizontale en fonction du taux de suppression appliqué dans les opérations de *DML*.

Le graphique suivant montre l'impact enregistré en fonction du taux de suppression que subit la table. Ce graphique peut être comparé avec la Figure 6-7 : Taux de remplissage de l'index lors d'opérations d'insertions ordonnées et de suppressions.

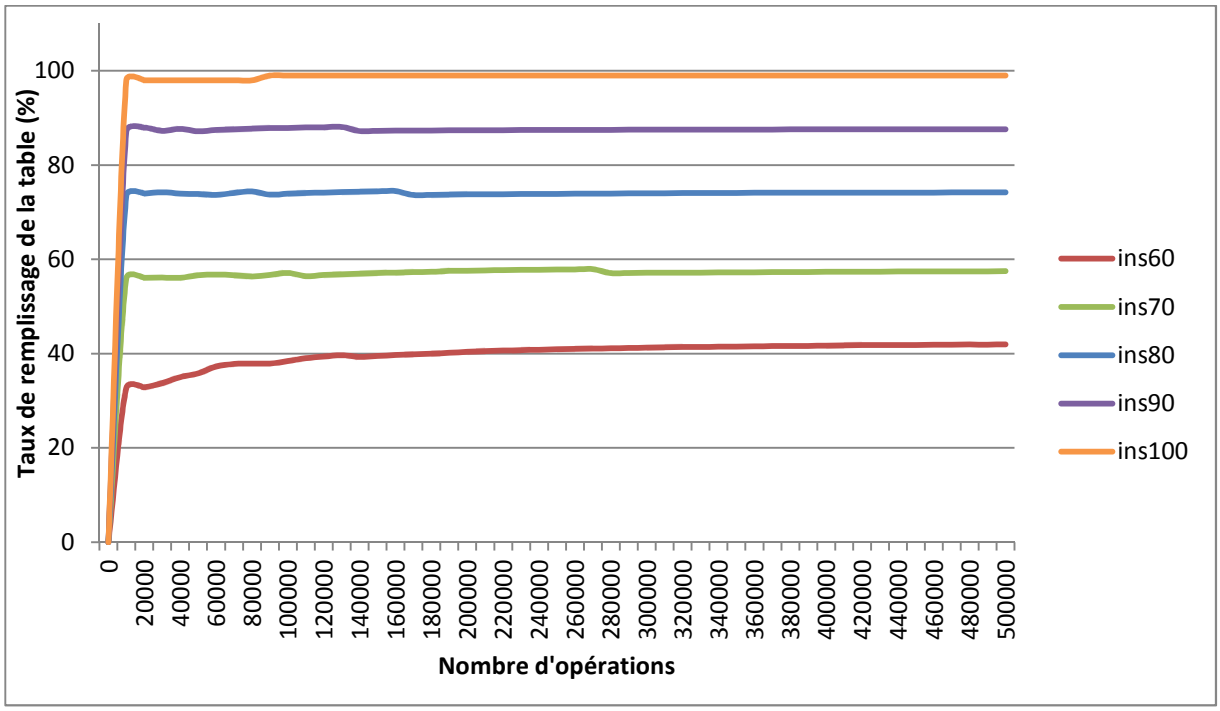


**Figure 7-5 : Taux de remplissage de l'index (insertions ordonnées et suppressions désordonnées)**

Les performances de la table ont également tendance à diminuer en fonction du taux de suppression. Dans ce cas de figure, il est préférable d'analyser la situation à partir de 15% de taux de suppression. Au-delà de 20% de suppressions dans les opérations *DML*, la perte de performance s'accentue davantage.

### Ajouts désordonnés et suppressions désordonnées

Le comportement du taux de remplissage de la table lors d'insertions combinées à des suppressions aléatoires font dramatiquement chuter le taux de remplissage de la table *IOT*. Quel que soit le taux de suppression, le taux de remplissage de la table est de 53%. Les raisons sont exactement les mêmes que celles évoquées dans l'ajout désordonné au sein d'une *IOT* (cf. section Ajout désordonné).

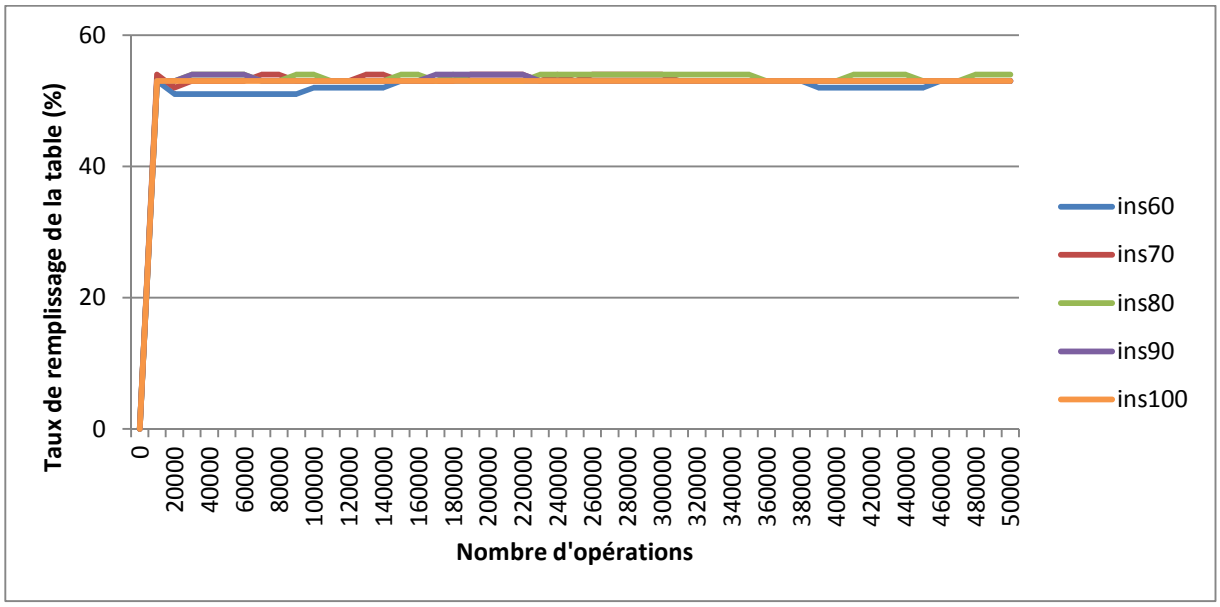


**Figure 7-6 : Taux de remplissage de l'index (insertions ordonnées et suppressions désordonnées)**

La figure ci-dessus montre la tendance à l'utilisation partielle de l'index dans une structure de type *IOT* qui subit des insertions aléatoires. Ce même phénomène se produit quelle que soit la taille d'un enregistrement.

## 7.6 Synthèse

L'analyse de la table *IOT* montre que la performance de celle-ci est liée à plusieurs facteurs. Il s'agit principalement de l'ordonnancement des clés d'enregistrement insérées (ordonnées ou aléatoires), de la taille de l'enregistrement et enfin, du taux d'opération de suppressions ou de mises à jour. Cette section décrit par un modèle mathématique les volumes de ce type de segments ainsi que le temps nécessaire pour y accéder en lecture et en écriture. Ensuite, il présente le comportement d'une table de type *IOT* à l'aide de plusieurs simulations. Ces exemples démontrent que la performance d'une table *IOT* est fortement liée à l'ordre des données qui lui sont injectées. En cas d'insertions aléatoires, la tendance est d'avoir un taux d'utilisation des blocs de 50%. Ce faible taux fait d'autant plus baisser les performances de cette structure.

# 8 Grille de comparaison

## 8.1 Préambule

L'ensemble des sections exposées dans ce travail permet de construire un modèle et de l'analyser. Il est maintenant intéressant d'identifier la structure la plus adaptée en fonction de certaines hypothèses, c'est-à-dire le type d'opérations, l'ordonnancement des clés d'enregistrement, la taille des blocs, les taux d'accès et les taux d'écriture. Il est également opportun d'analyser de quelle situation une structure tire profit ainsi que les opérations qui lui sont préjudiciables. Quelle est le meilleur choix à réaliser en termes d'occupation de disque ? Qu'en est-il de la taille des blocs ? Que doit attendre un système en termes de vitesse et dans quels cas apparait le phénomène de fragmentation ?

Cette section a pour objectif de synthétiser l'ensemble des informations collectées afin d'avoir une vision globale des performances sous certaines hypothèses.

## 8.2 Taux d'occupation du disque

Le tableau ci-dessous s'attarde principalement sur les taux d'occupation de l'espace dans les différents cas présentés plus haut.

La première partie du tableau présente les taux d'occupation des tables sur le disque. La seconde, quant à elle, présente le taux d'organisation séquentielle des blocs par rapport aux valeurs qu'ils contiennent.

| | *HOT* avec index | *IOT* |
|---|---|---|
| Table en lecture seule | Bon | Excellent |
| Insertions valeurs croissantes | Bon | Excellent |
| Insertions valeurs décroissantes | *HOT* : Bon<br>Index : Mauvais | Excellent |
| Insertions valeurs aléatoires | *HOT* : Excellent<br>Index : Bon | Mauvais |
| Opérations multiples à valeurs croissantes | *HOT* : Bon<br>Index : Bon avec proportion d'insertions supérieure à 80% | Excellent avec proportion d'insertions supérieure à 80% |
| Opérations multiples à valeurs aléatoires | *HOT* : Bon<br>Index : Moyen à Bon | Mauvais |
| Table en lecture seule | Excellent | Excellent |

| | | |
|---|---|---|
| Insertions valeurs croissantes | Excellent | Excellent |
| Insertions valeurs aléatoires | *HOT* : Mauvais<br><br>Index : Bon | Bon |
| Opérations multiples à valeurs croissantes | *HOT* : Bon avec proportion d'insertions supérieure à 80%<br><br>Index : Excellent | Excellent |
| Opérations multiples à valeurs aléatoires | *HOT* : Mauvais<br><br>Index : Moyen à Bon | Mauvais |

**Tableau 1 : Comparaison du taux d'occupation de l'espace disque et du taux de fragmentation logique entre une table *HOT* et une table *IOT***

## 8.3 Taille des blocs

De manière générale, augmenter la taille des blocs entraîne une diminution des divisions de type 50/50 (opération lourde) et a tendance à augmenter les performances d'accès et d'insertions.

Globalement, une taille de bloc plus élevée augmente les performances du *SGBD*. En effet, cela diminue le taux de migrations et de chaînages des enregistrements, tout comme le taux de divisions et la surcharge liée à la gestion des blocs.

Cependant, une taille de bloc élevée a également tendance à diminuer la libération des blocs dans des systèmes à haut taux de suppression.

## 8.4 Vitesse d'accès

Les méthodes d'accès et, donc, les vitesses d'accès varient en fonction du type d'objets ainsi que de leur état. Le tableau suivant propose une appréciation des temps d'accès en fonction de ces facteurs.

| | *HOT* avec index | *IOT* |
|---|---|---|
| Lecture d'une clé (ordonné) | Bon | Excellent |
| Lecture séquence (ordonné) | Bon | Excellent |
| Lecture d'une clé (désordonné) | Bon | Excellent |
| Lecture séquence (désordonné) | Moyen | Mauvais |

**Tableau 2 : Comparaison des vitesses d'accès entre une table *HOT* et une table *IOT***

## 8.5 Vitesse en écriture

L'écriture au sein de ces structures varie en fonction de plusieurs critères :

- La taille d'un enregistrement ;
- L'ordre des données écrites ;
- La construction de l'index après l'insertion (avec une table *HOT*).

La structure *IOT* est fortement affectée lorsque son taux de divisions devient élevé. Lors d'insertions désordonnées, cette structure augmente significativement son taux de divisions. La structure *HOT* est donc plus performante que l'*IOT* pour les insertions désordonnées. La table *HOT* bénéficie également de la possibilité de désactiver son index lors d'un chargement, puis de le reconstruire par la suite de manière asynchrone. Ceci permet un chargement extrêmement rapide des données au sein de la table *HOT*. Enfin, une suppression demande deux fois plus de L/E pour une table *HOT* que pour une table *IOT*.

| | *HOT* avec index | *IOT* |
|---|---|---|
| Ecriture (ordonnée) | Bon | Excellent |
| Ecriture (désordonnée) | Bon | Mauvais |
| Ecriture sans index | Excellent | S.O. |
| Modification | Bon | Moyen |
| Suppression | Moyen | Excellent |

**Tableau 3 : Comparaison des vitesses d'écriture entre une table *HOT* et une table *IOT***

## 8.6 Dégradation de l'espace occupé

La dégradation de l'espace est liée à l'insertion de données de manière désordonnée ou à des opérations *DML* sur les tables. L'insertion de données ordonnées et croissantes maximise le taux de remplissage des deux structures. La table *IOT* est toujours plus performante que la structure *HOT* lors d'ajout ordonné. Par contre, son taux d'occupation diminue nettement lors d'insertions aléatoires. La fragmentation engendrée par des suppressions n'est pas corrigée par le moteur Oracle puisque celui-ci ne fusionne pas les blocs sans une intervention des *DBA*. Par conséquent, la suppression est un élément peu optimum pour chacune des deux structures.

| | *HOT* avec index | *IOT* |
|---|---|---|
| Ecritures (ordonnées croissantes) | Table : Bon<br>Index : Bon | Excellent |
| Ecritures (ordonnées décroissantes) | Table : Bon<br>Index : Mauvais | Excellent |
| Ecritures (désordonnées) | Table : Bon<br>Index : Moyen | Mauvais |
| Modifications | Bon | Bon |
| Suppressions | Mauvais | Mauvais |

**Tableau 4 : Comparaison des taux de remplissage d'une table *HOT* et d'une table *IOT***

## 8.7 Dégradation des accès par clé

Cette dégradation est liée au phénomène de migration et de chaînage. La table *IOT* n'est pas affectée par la migration. La structure *HOT,* quant à elle, doit toujours réaliser une lecture complémentaire pour accéder à l'enregistrement. Cette dégradation est fortement liée à la mise à jour des enregistrements. La table *IOT* apparait plus efficace pour ce type d'accès grâce, d'une part, à sa structure et, d'autre part, parce qu'elle ne réalise pas de migration de ses enregistrements. Cela est plus coûteux lors des mises à jour, mais permet un accès plus rapide par la suite étant donné qu'il est possible d'accéder directement à l'information.

## 8.8 Dégradation des accès séquentiels

La dégradation lors d'un accès à une séquence d'enregistrements est l'élément le plus évident et le plus pénalisant. En effet, l'accès séquentiel est impacté par de nombreux éléments tels que le taux de remplissage, le nombre de segments contigus et le taux de rupture des données au sein des blocs qui composent le segment. Les sections et 7 montrent combien le chargement des enregistrements a un impact important sur les performances des accès séquentiels. Chaque division qui n'est pas de type 90/10 induit une rupture dans une lecture séquentielle des données. De plus, un haut taux de modifications (modifications et suppressions) entraîne un haut taux de recyclage des blocs et induit, également, un haut taux de ruptures dans les blocs de données, et ce, même lors d'insertions de valeurs ordonnées croissantes. Le taux de modifications de 20% provoque un impact significatif sur une table, et demandera une analyse de l'état de la structure. Le cas le plus défavorable est la lecture séquentielle d'une table *HOT* dont les enregistrements ne sont pas organisés sur l'index : Le coût de cette lecture peut être décuplé. Le cas le plus favorable est la lecture d'une table *IOT* qui reçoit des valeurs ordonnées (croissantes ou décroissantes). Les tables *HOT* et *IOT* subissent une dégradation de performances lente lors d'une situation comportant des insertions ordonnées. Dans le cas contraire, la structure *HOT* devient rapidement inefficace. La structure *IOT,* quant à elle, subit une perte de performance mais continue à disposer d'un meilleur taux d'accès que la table *HOT*.

## 8.9 Synthèse

Cette section a pour objectif de réaliser une synthèse de l'ensemble des éléments collectés dans les pages précédentes. Sous certaines hypothèses, force est de constater que certaines caractéristiques s'effondrent : une taille sur disque s'accroit de manière significative, ou encore les temps différents d'accès qui se dégradent.

La structure *IOT* offre de meilleures performances en accès à condition qu'elles soient chargées avec un ensemble de données ordonnées (qu'elles soient croissantes ou décroissantes). A contrario, plus la taille de l'enregistrement est grande, plus le taux de divisions est élevé lors d'opérations d'insertions. Par conséquent, cette table est nettement moins performante lors de chargement ou de modification d'informations. Son mode de division particulier lui confère également un net désavantage en cas d'insertions aléatoires, poussant sa structure à avoir un taux de remplissage proche des 50%. Dès lors, le parcours complet de cette table devient deux fois plus coûteux en termes de L/E. Le taux de perte de performances en lecture séquentielle est doublé dans le cas le moins favorable.

La structure *HOT*, quant à elle, est plus efficiente lors d'un chargement ordonné croissant. La taille de la clé d'index influence le taux de division au sein de l'index. La taille de l'ensemble de l'enregistrement n'a pas d'effet sur les phénomènes de divisions. La performance lors d'une opération d'insertion est meilleure que celle de la table *IOT*. L'index de la table *HOT* obtient un taux

de remplissage idéal lors d'un chargement d'enregistrements ordonnés et croissants. En cas d'insertions sur des valeurs de clés aléatoires, ce dernier oscille entre 65 et 75% d'utilisation, en tendant vers 70%. Le taux de chargement devient catastrophique en cas d'insertions ordonnées sur des valeurs décroissantes : le taux devient alors de 50%. Le taux de performances de la lecture séquentielle de la table *HOT* est fortement lié avec l'organisation des données au sein de celle-ci, et ce, quel que soit le taux de chargement de l'index. Les performances de lectures séquentielles d'une table *HOT* affichent des performances 11 fois moins bonnes en cas d'insertions aléatoires.

Enfin, lors d'insertions combinées à des manipulations de DML (suppressions ou modifications), la table *IOT* semble légèrement plus performante lors d'insertions ordonnées et demande moins rapidement une réorganisation de sa structure. En revanche, ses performances deviennent mauvaises lorsque les clés d'index ne sont pas ordonnées. La table *HOT* profite de performances relativement bonnes dans les deux situations (insertions ordonnées et désordonnées). Les deux structures voient leurs performances respectives se dégrader significativement en dessous de 80% de taux d'insertion.

# Conclusion

Ce mémoire avait pour objectif d'exposer une problématique liée à l'utilisation d'un système de base de données relationnelles Oracle : le phénomène de dégradation des performances.

Pour atteindre cet objectif, nous avons réalisé une analyse des mécanismes de fonctionnement d'Oracle. Celle-ci a permis de mettre en avant le fait que l'organisation des structures logiques contenant les données pouvait être sujette à des dégradations de performance au niveau du temps d'accès. Partant de ce constat, nous avons choisi de modéliser trois types d'objets couramment utilisés par les bases de données : la structure Heap Organized Table (*HOT*), l'index de celle-ci et la structure Index Organized Table (*IOT*).

Nous nous sommes penchés sur les phénomènes qui pouvaient potentiellement induire une dégradation des performances du système. Nous avons défini plusieurs cas d'utilisation des structures (insertions, suppressions, modifications) dans diverses situations telles que les systèmes dédiés, partagés ou fortement partagés.

À partir de ces cas de figure, nous avons déterminé les différents cas de fragmentation pouvant affecter la dégradation des performances d'une base de données.

Ensuite, nous avons défini les opérations classiques de lecture, d'insertion, de suppression et de mise à jour de ces structures au travers de modèles mathématiques. Le but était de connaître et de quantifier l'impact des opérations utilisées sur les structures précédemment citées en termes de dégradation de performances.

Enfin, nous avons simulé l'utilisation de ces structures dans les différents cas de figure susmentionnés afin de comparer leur comportement réel avec les modèles mathématiques définis tout au long de ce mémoire.

À l'issue de cette modélisation, nous avons analysé les utilisations optimales des différentes structures (*HOT*, Index et *IOT*) afin de connaître les applications qui permettaient de diminuer les pertes de performances du système. De plus, nous avons présenté les cas d'utilisations les moins favorables pour chacune de ces structures. Enfin, sous certaines hypothèses, nous constatons une perte de performance régulière. Pour cette raison, nous avons défini les seuils à partir desquels l'information devait être réorganisée afin de contourner cette problématique.

La plupart des simulations ont fourni un résultat tel qu'attendu au vu de la modélisation. Cependant, deux des scénarios ont présenté des comportements différents de ceux escomptés. Il s'agit des deux cas d'insertions aléatoires au sein d'une structure *IOT*. Ces divergences par rapport au modèle établi ont conduit à une révision du modèle de division au sein de la structure *IOT*. Le nouveau modèle a permis d'expliquer cet écart et de mieux comprendre le fonctionnement de cette structure.

A l'issue de ce mémoire, nous en connaissons davantage sur les phénomènes de fragmentation et sur leur impact sur les performances du système.

Ce travail m'a permis de constater que les informations trouvées sur internet (forums d'experts Oracle et de support) ne sont pas toujours de premières qualités et que les explications sont parfois basées sur une ancienne version du SGDB, explications parfois mal comprises et résumées en de dangereuses généralités.

# Bibliographie


[1] KYTE, T. *Expert Oracle Database Architecture, Oracle Database 9i, 10g and 11g Programming Techniques and Solutions,* 2nd ed. Apress, U.S.A, 2010.

[2] LEWIS, J. *The Clustering Factor* in *Cost-Based Oracle Fundamentals,* Apress, U.S.A, 2006, 87-115.

[3] HAINAUT, J.-L. *Implémentation des structures de données* in *Bases de données. Concepts, utilisation et développement,* 2e ed. Dunod, 2012.

[4] ORACLE® *How to identify a Hot block within de Database buffer cache* ? **URL** : https://support.oracle.com, ID 163424.1 (consulté le 10/06/2012).

[5] ORACLE® *How does the Index bloc splitting mechanism work for B-Tree indexes ?* **URL** : https://support.oracle.com, ID 183612.1 (consulté le 12/07/2012).

[6] ORACLE® *How to identify, avoid and eliminate chained and migrated rows ?* **URL** : https://support.oracle.com, ID 746778.1 (consulté le 04/05/2013).

[7] ORACLE® *How B-trees indexes are maintained ?* **URL** : https://support.oracle.com, ID 30405.1 (consulté le 13/07/2012).

[8] ORACLE® *The clustering factor.* **URL** : https://support.oracle.com, ID 39836.1 (consulté le 12/07/2012).

[9] ORACLE® *Wat is an Index Organized Table (IOT) and when to use an IOT ?* **URL** : https://support.oracle.com, ID 176041.1 (consulté le 30/06/2013).

[10] ZAHN, M. *The secrets of Oracle row chaining and migration.* **URL :** http://www.akadia.com/services/ora_chained_rows.html, Septembre 2007 (consulté le 18/02/2013).

[11] ENSOR, D. *Optimal Physical Database Design for Oracle8i.* **URL :** http://www.dba-oracle.com/art_dbazine_ensor_optimizing_oracle_physical_design.htm (consulté le 18/06/2013).

[12] FOGEL, S. *Oracle® Database Administrator's Guide*. **URL** : http://docs.oracle.com/cd/B28359_01/server.111/b28310.pdf, March 2008 (consulté le 02/07/2012).

[13] AULT, *M. PCTFREE and PCTUSED Tips.* **URL :** http://www.dba-oracle.com/oracle_tips_PCTFREE_PCTUSED.htm, August 2004 (consulté le 24/08/2013).

[14] FOOT, R. Indexes and Small Tables Part IV (treefingers). **URL**: http://richardfoote.wordpress.com/category/root-index-block/, May 2009 (consulté le 23/08/2013).

[15] HARRISON, G. *Oracle® Performance Survival Guide. A Systematic Approach to Database Optimization*, Pearson, Boston, October 2009.